60189863.031600

Putty - 8045

Association in Rhino

An Association represents a relationship between an Item and other Item(s) in the same or different Category. For example, the user account Item(s) that are related to a group account Item with selector "bar" could be represented by an Association. The identifying parameters for each Association are a parent Category, selector of an Item in that Category (parent Item) and a child Category. An Association represents an 1-to-n relationship. A 1-to-1 relationship is handled as a special case of 1-to-n relationship.

Association derives from Category and supports the same client mechanisms, using CategoryListener interface, for obtaining dynamic information about the Item(s) in an Association instance. The list of Item(s) in an Association is of the same type as indicated by the (selector of) child Category. An Association can have a set of Attributes that apply to all Item(s) in that relationship. The Association class provides API to support monitoring of Item(s) and Attribute(s) in the specific relationship and notification of current state and state changes to interested clients.

If an Item with the specified parent selector does not exist, the Association will monitor the parent Category for the addition of an Item with that selector. When such an Item is added, the subclasses are notified and subclasses can do whatever is necessary to determine the child Item(s) and Association Attribute(s) and monitor the system for future changes.

If an Item with the specified parent selector existed, but is deleted while a client is monitoring the Association, all Items are removed from the Association and the Association continues monitoring of the parent Category for the addition of an Item with that parent selector.

The concept of an Association is implemented in Java at the client-side and in C++ at the server-side. An application developer writes the logic to determine the set of Item(s) related to a parent Item and Association Attribute(s) and to monitor the system for any changes to the Item(s) and Association Attribute(s) in C++, using the server-side API. The application-specific clients, such as specific Tasks, ItemViews and TreeView are written in Java and use the client-side API to obtain information about Association(s) and Item(s).

Implementing Association(s) on the server-side: C++

The Association class and three subclasses **ComputedAssoc**, **ChildAttrAssoc** and **ParentAttrAssoc** provide support for implementing Association(s) on the server-side.

Association

The most basic support for specifying association is provided by the Association class. This class monitors the parent Item and subclasses are responsible for determining the child Item(s) that added, deleted and changed. The Association class (versus derived classes) is used only when sub-classes require logic very specific to the application to determine when items are added, changed and removed from the association. For example, the relationship may be calculated by queries on a database based on the parent Selector. The derived classes ComputedAssoc, ChildAttrAssoc and ParentAttrAssoc is used when the data requiring for computing the relationship is contained within the Item(s) in the Category(s).

Infrastructure Support for Association

```
// Provided by infrastructure
class Category : public AttrBundle {
    ...
    virtual void addItem(const& Item item);
    virtual void removeItem(const& Item item);
    virtual void changeItem(const& Item oldItem, const& Item newItem);
    ...
};

// Provided by infrastructure
class Association : public Category {
    ...
    Association(Category& parentCategory, const String& parentSelector,
                Category& childCategory);

    // Association interacts with Category with selector parentCategory
    // and calls the following methods when the parent Item with
    // selector parentSelector is added, changed or removed.
    virtual void parentAdded(const Item& item);
    virtual void parentChanged(const Item& oldItem, const Item& newItem);
    virtual void parentRemoved();
    ...
};
```

Using Association

```
// Written by application developer
class MachinesAssocWithCluster : public Association {
    MachinesAssocWithCluster(Category& parentCategory,
                             const String& parentSelector,
                             Category& childCategory) :
        Association(parentCategory, parentSelector, childCategory);
        virtual void parentAdded(const Item& item);
        virtual void parentChanged(const Item& oldItem,
                                   const Item& newItem);
};
```

The Association class provides trivial implementations of `Association::parentAdded()` and `Association::parentChanged()`. Subclasses can override `Association::parentAdded()` to, for example, register for notifications about Items in the child Category. Subclasses can override `Association::parentChanged()` to do whatever is necessary to keep the list of child Item(s) and Association Attribute(s) up-to-date. The Association class implements `Association::parentRemoved()` to remove all Item(s) from its list and notify removal of Item(s) to registered listeners. Most of the rules of subclass interaction with the Category base class apply. The differences are:

- `Category::startMonitor()` is overriden by the Association base class to start monitoring of the parent Item. The Association class will call `Association::parentAdded()` if the parent Item is

P1

determined to exist upon start of monitoring or added later. Subclasses of Association should not call `Category::addItem()`, `Category::changeItem()`, `Category::removeItem()`, `Category::replaceItemList()`, `AttrBundle::setAttr()` prior to the call to `Association::parentAdded()`.

- The selector of the Association instance is not the same as the type Item(s) in its list. Item(s) in the list are of type corresponding to the selector of the child Category.

Typically a subclass makes zero or more `Category::addItem()` and `AttrBundle::setAttr()` calls, followed by a `Category::endExists()` call followed by zero or more `Category::addItem()`, `Category::changeItem()`, `Category::removeItem()` and `Category::setAttr()` calls. The `Category::endExists()` call signals that the subclass has communicated the entire set of Item(s) and Association attributes discovered in the system to the Category base class.

The anticipated use of information in Association is by the client side code. Clients can access Association information using the CategoryListener interface in the same manner as they would obtain information from a Category. The only difference is the call to obtain the handle to an Association. This is covered in detail in the Obtaining information about Association(s) on the client-side section. The information in that section can be applied to server-side components requiring information from an Association via the C++ CategoryListener API.

ComputedAssoc

This is the base class for deriving classes to represent relationships that can be computed from values of Attribute(s) of the monitored parent and child Item(s). This class monitors the parent Item from the parent Category and Item(s) in the child Category that are potential children. The Item(s) that are potential children are indicated by a NotificationFilter that comes in effect when the parent Item is detected by Association. The NotificationFilter can be changed when the parent Item changes or at any arbitrary time.

Infrastructure Support for ComputedAssoc

```
// Provided by infrastructure
class ComputedAssoc : public Association {
    ...
    // Monitoring of potential child Item(s)
    virtual void childCategoryItemAdded(const Item& item);
    virtual void childCategoryItemChanged(const Item& oldItem,
                                          const Item& newItem);
    virtual void childCategoryItemRemoved(const String& selector);
    ...
    // Set NotificationFilter to indicate potential child Item(s)
    virtual NotificationFilter*
            createAddedChildNotificationFilter(const Item& parentItem);
    virtual NotificationFilter*
            createChangedChildNotificationFilter(const Item& oldItem,
                                                 const Item& newItem);
    virtual void adoptAndReplaceChildNotificationFilter(NotificationFilter*
                                                        filter);
    ...
    //
    ...
    virtual bool isChild(const Item& potentialChildItem) = 0;
};
```

P2

Subclasses must override the `ComputedAssoc::isChild()` method to provide logic that determines if the Item of the child Category passed to `ComputedAssoc::isChild()` is a child of the parent Item.

This class marks all Item(s) of the child Category as potential child Item(s). This class calls `ComputedAssoc::isChild()` anytime there is a change to the parent or the monitored child Item(s) and forwards notifications about changes in its list of child Item(s) to the listeners on this Association. When the parent Item changes, it gets the current list of child Item(s) and checks if any child Item(s) need to be added/removed from its list based on the computation performed by `ComputedAssoc::isChild()`. When an Item of the child Category is added, removed or changed, it checks if the Item should be added/removed/updated in its list of child Item(s) based on isChild().

For example, the user account Item(s) belonging to a particular group account Item could be st. the user account Item(s) that have the same value of Attribute with key *uid* as the corresponding Attribute in the group account Item. The following code can be used to model this relationship.

Using ComputedAssoc

```
// Provided by application developer
class UsersAssocWithGroup : public ComputedAssoc {
   ...
   bool isChild(const Item& parentItem, const Item& childItem) {
       if (childItem.getAttr("uid").stringValue() ==
           parentItem.getAttr("uid").stringValue()) {
             return true;
           } else {
             return false;
           }
   }
   ...
}
```

Sub-classes can further fine-tune behaviour by overriding the methods that are called upon notifications related to the parent and child Item(s). For example, subclass ChildAttrAssoc overrides parentChanged() to turn off computation of the list of child Item(s) when the parent Item changes and only does so on changes in the Item(s) of the child Category.

ChildAttrAssoc

This is the base class for deriving classes to represent relationships in which the child Item(s) store the selectors of one or more parent Item(s) as part of its Attributes.

Subclasses provide the Attribute's key in the constructors. The ParentAttrAssoc class monitors the parent and child Item(s), keeps the list of child Item(s) current and notifies listeners of changes to the list of child Item(s).

ParentAttrAssoc can determine the Item(s) of child category that belong to this Association in two ways depending on the constructor that is used. One constructor is used when there is a 1-to-1 relationship from a child Item to Item(s) of the parent Category. Subclasses specify the Attribute key of the child Item that holds the parent Item selector. Another constructor is used when there is a 1-to-n relationship from a child Item to Item(s) of the parent Category. This is based on the recommended format for representing arrays of values in an Item. Subclasses specify the value of the key of the child Item that holds the number of parent Item selectors and the base name of the Attribute keys which hold the selectors themselves.

For example, if ClustersAssocWithMachine is a relationship where an Item of type ClusterCategory has an array of selectors of Item(s) of MachineCategory, NUM_MACHINES could be the key of ClusterCategory Item(s) that holds number of values in the array. The base name of the attribute keys could be MACHINE. ParentAttrAssoc will monitor Item(s) corresponding to the list of selectors in Attributes of the child ClusterCategory Item(s) with keys CLUSTER0, CLUSTER1, ... CLUSTER<*NUM_CLUSTERS* - 1>.

ChildAttrAssoc keeps the list of child Item(s) up-to-date based on parent/child changes.

Infrastructure Support for ChildAttrAssoc

```
// Provided by infrastructure
class ChildAttrAssoc : public ComputedAssoc {
    ...
    // Constructor.  This version is used for 1-to-n relationships and
    // takes two attribute names.
    // "childAttrNumKeys" is the attribute key which holds the number of
    // parent Item selectors, and "childAttrKey" is the base name of
    // the attribute keys which hold the selectors themselves.
    ChildAttrAssoc(Category& parentCategory,
                   const String& parentSelector,
                   Category& childCategory,
                   const String& childAttrNumKeys,
                   const String& childAttrKey);

    // Constructor.  This version is used for 1-to-1 relationships and
    // takes one attribute name which is the attribute key holding the
    // parent Item selector.
    ChildAttrAssoc(Category& parentCategory,
                   const String& parentSelector,
                   Category& childCategory,
                   const String& childAttrKey);
    ...
}
```

Typically, subclasses only need to call the ChildAttrAssoc constructor with the application specific values of the Attribute keys and do not need to override any methods. For example, if a user account Item has an attribute with key "groupSelector" that refers to the parent group account Item, the following code can be used to model this relationship.

Using ChildAttrAssoc

```
// Provided by application developer
class UsersAssocWithGroup : public ChildAttrAssoc {
    UsersAssocWithGroup(Category& parentCategory,
                        const String& parentSelector,
                        Category& childCategory) :
        ChildAttrAssoc(parentCategory, parentSelector, childCategory,
                       "groupSelector") {
    }
};
```

ParentAttrAssoc

This is the base class for deriving classes to represent relationships in which the parent Item stores the selectors of one or more child Item(s) as part of its Attributes.

Subclasses provide the Attribute's key in the constructors. The ParentAttrAssoc class monitors the parent and child Item(s), keeps the list of child Item(s) current and notifies listeners of changes to the list of child Item(s).

ParentAttrAssoc can determine the Item(s) of child category that belong to this Association in t depending on the constructor that is used. One constructor is used when there is a 1-to-1 relatio from a parent Item to Item(s) of the child Category. Subclasses specify the Attribute key of the parent Item that holds the child Item selector. Another constructor is used when there is a 1-to-n relationship from a parent Item to Item(s) of the child Category. This is based on the recommended format for representing arrays of values in an Item. Subclasses specify the value of the key of the parent Item that holds the number of child Item selectors and the base name of the Attribute keys which hold the selectors themselves.

For example, if ClustersAssocWithMachine is a relationship where an Item of type MachineCategory has an array of selectors of Item(s) of ClusterCategory, NUM_CLUSTERS could be the key of MachineCategory Item(s) that holds number of values in the array. The base name of the attribute keys could be CLUSTER. ParentAttrAssoc will monitor Item(s) corresponding to the list of selectors in Attributes of the parent MachineCategory Item with keys CLUSTER0, CLUSTER1, ... CLUSTER<*NUM_CLUSTERS* - 1>.

ParentAttrAssoc keeps the list of child Item(s) up-to-date based on parent/child changes.

Infrastructure Support for ParentAttrAssoc

```
// Provided by infrastructure
class ParentAttrAssoc : public ComputedAssoc {
    ...
    // Constructor.  This version is used for 1-to-n relationships and
    // takes two attribute names.
    // "parentAttrNumKeys" is the attribute key which holds the number of
    // child Item selectors, and "parentAttrKey" is the base name of
    // the attribute keys which hold the selectors themselves.
    ParentAttrAssoc(Category& parentCategory,
                    const String& parentSelector,
                    Category& childCategory,
                    const String& parentAttrNumKeys,
                    const String& parentAttrKey);

    // Constructor.  This version is used for 1-to-1 relationships and
    // takes one attribute name which is the attribute key holding the
    // child Item selector.
    ParentAttrAssoc(Category& parentCategory,
                    const String& parentSelector,
                    Category& childCategory,
                    const String& parentAttrKey);
    ...
```

P3

```
};
```

Typically, subclasses only need to call the ParentAttrAssoc constructor with the application specific values of the Attribute keys and do not need to override any methods. For example, if a group account Item has an attribute with key "NUM_USERS" that specifies the number of user account Item(s) belonging to it, USER0, USER1 ... USER<*NUM_USERS* - 1> would be the keys of the actual selector values, the following code can be used to model this relationship.

```
// Provided by application developer
class UsersAssocWithGroup : public ParentAttrAssoc {
    UsersAssocWithGroup(Category& parentCategory,
                        const String& parentSelector,
                        Category& childCategory) :
        ParentAttrAssoc(parentCategory, parentSelector, childCategory,
                        "NUM_USERS", "USER") {
    }
};
```

Plug-in an Association into the Rhino Infrastructure

AssocFactory is the factory class for Association objects. AssocFactory methods are used by the Association Service (similar to the Category Service described in sysadmd(1M)) to fulfill requests from remote clients. They can also be used by any server-side components that require information from an Association. Association subclasses use the macros defined by AssocFactory.h.

To make information about an Association between any Item of type *parentCategorySelector* and Item(s) of type *childCategorySelector* available to the rest of the system the following steps are required:

1. Information about the Category(s) *parentCategorySelector* and *childCategorySelector* should be available as detailed in the document Item and Category in Rhino.
2. Implement a subclass of Association called <parentCategorySelector>AssocWith<childCategorySelector>.
    1. The subclass must have a constructor with the signature (Category& parentCategory, const String& parentSelector, Category&). Typically, this calls the corresponding Association constructor or any Association subclass constructor. *parentSelector* is the name of the parent Item in a Category instance *parentCategory* corresponding to *parentCategorySelector* for which related Item(s) from Category instance *childCategory* corresponding to *childCategorySelector* are to be determined.
    2. Use the convenience macro SaASSOC_REF_DECL, provided by AssocFactory.h, in the header file to provide declaration for the routines used by the Association Service for obtaining Association instances.
    3. In most cases, only one instance of a particular subclass of Association should exist for a given "parentCategory", "parentSelector", "childCategory" combination in an address space. To enforce this, subclasses should protect their constructors and use the convenience macro SaASSOC_FRIEND_DEF, provided by AssocFactory.h, in the class declaration in the header file. This allows AssocFactory access to the protected constructors.
    4. Use the convenience macro SaASSOC_REF_DEF, provided by AssocFactory.h, in the c++ file to provide the definition for the routines used by the Association Service for obtaining Association instances. This in turn will use the mandatory constructor described.
3. Create a library called <parentCategorySelector>AssocWith<childCategorySelector>.so.
4. Install it in /usr/sysadm/association/

The above steps will allow clients to obtain the Association instance to determine the relationship between any Item of type *parentCategorySelector* and Item(s) of type *childCategorySelector*.

Obtaining information about Association(s) on the client-side: Java

All application-specific entities are instances of Item. Further, all application-specific associatio[illegible] instances of Association. No subclassing is required by the developer of specific application. T[illegible] in obtaining information about Association(s) and Item(s) are:

- Obtaining a handle to an Association instance.
- Obtaining information about Item instances.

Obtaining a handle to an Association instance

Association instances are obtained via a HostContext object. When writing Task UI interface, a HostContext object will be available for you from the Task infrastructure. The same applies to writing an ItemView etc,.

Internally, the HostContext object is obtained when a user successfully logs in to a server machine.

If the HostContext object is *hostContext*, then a client can obtain a handle to an Association representing the user account Item(s) belonging to a parent group account Item called *foo* by using the following code:

```
Association assoc =
    hostContext.getCategory("GroupAccountCategory", "foo",
                            "UserAccountCategory");
```

This is an asynchronous call that returns an handle to the Association before it receives a respo[illegible]m the server. The client can use this handle to add CategoryListener instances for obtaining infon[illegible] an error is encountered in communication with the server or loading the specific Association insta[illegible] requested, this is handled as a fatal connection error by the infrastructure and the client will exit after the error message is acknowledged by a user.

Obtaining information about Items

All details of Obtaining information about Items from a category apply to obtaining information from an Association.

P4

Ph#y-8046

Item and Category in Rhino

This document describes how Category(s) and Item(s) are supported in the Rhino infrastructure. The first section discusses the underlying concepts behind Category(s) and Item(s). The second section presents the API for implementing Category(s) and Item(s) at the server side. The last section presents the API to obtain dynamic information about a specific Category and Item at the client side. Typical clients of Category(s) and Item(s) include Rhino UIs, such as Tasks and ItemViews.

Underlying Concepts

The fundamental data types used in Rhino are the **Attribute** class and the **AttrBundle** class. An Attribute is a typed key-value pair. The different Attribute types supported are boolean, long, double and string. AttrBundle is an aggregation of Attribute(s). Each AttrBundle instance has two string fields representing the *type* and *selector*. Subclasses of AttrBundle intepret these fields in different ways.

An Item is a subclass of AttrBundle. The type of an Item instance is the name (selector) of the Category instance that it belongs to. The selector of an Item instance is the unique name of the instance within the Category instance. For example, a user account Item instance can have selector *foo* within the type *UserAccountCategory* and the following Attributes:

```
        string, userName, foo
        long, userId, 3944
```

Arrays of values can be represented in an Item as follows:

1. One attribute specifies the number of values in the array.
2. Each value is accessed by appending a number to a prefix for the series of values.

For example, if an Item of type group account has an array of Strings representing names of users that use its group id, this would by represented by an Attribute with key, for example, NUM_USERS that specifies the number of values in the array and a key, for example, USER as the prefix to use. USER0, USER1 ... USER<*NUM_USERS* - 1> would be the Attribute keys of the actual String values. For example, NUM_USERS = 3, USER0 = foo, USER1 = bar, USER2 = baz.

This is the format that the Association mechanism relies on for parameters referring to the selectors of Item(s) to monitor the relationship between Item(s).

A Category class has a collection of Items. Category is also a subclass of AttrBundle. The type of a Category instance is the constant value *Category*. The selector is the unique name of the Category instance within the system. For example, the user account Category instance would have selector *UserAccountCategory*. A Category can have a set of Attributes that apply to all Item(s) of that type. For example, *UserAccountCategory* instance can have Attributes that store information about whether shadow passwords are in use by a system. Category classes provide API to support monitoring of the Item(s) and Attribute(s) of the specific type and notification of current state and state changes to interested clients.

Clients interested in information about a Category instance do so via a **CategoryListener interface**. A CategoryListener instance can be registered with a Category for notifications. Upon registration, the CategoryListener instance receives information about the current state of the Category instance. If the state of the Category changes the CategoryListener will receive information about the changes as they occur. Information from Category can be obtained at several granularities.

All client-server communication is asynchronous so that the UI can be responsive to user input and not block waiting for completion of a request to the server. Asynchronous nature is achieved by using a callback model.

The concepts of Category, Item and CategoryListener are implemented in Java at the client-side and in C++ at the server-side. An application developer writes the logic to determine the set of Item(s) (of a specific type) and Category Attribute(s) and to monitor the system for any changes to the Item(s) and Category Attribute(s) in C++, using the server-side API. The application-specific clients, such as specific Tasks or ItemViews are written in Java and use the client-side API to obtain information about Category(s) and Item(s).

Implementing Category(s) and Item(s) on the server-side: C++

All application-specific entities are instances of Items. No subclassing is required. A Category class needs to be subclassed. An instance of the subclass performs application-specific operations to obtain the state of the system and to inform the Category base class of any changes to the state. For example, the *UserAccountCategory* instance (of Category) would read and monitor the passwd files or NIS maps to monitor user account Item(s) to obtain the current state and detect changes.

When the first CategoryListener is added to a Category instance, the Category base class calls `Category::startMonitor()`. Category subclasses must override this method to do whatever is necessary to discover existing Item(s) and monitor Item(s) of the specific type. Information about all Item(s) that exist at the time `Category::startMonitor()` is called should be communicated to the Category base class via `Category::addItem()` calls. Information about Category attributes should be communicated by the subclass via `AttrBundle::setAttr()`. The end of the Item(s) and Catego Attribute(s) that exist when `Category::startMonitor()` is called should be communicated to Category base class via an `Category::endExists()` call. Any future addition, removal of Item(s), well as changes to the Item(s) should be communicated to the Category base class via `Category::addItem()`, `Category::removeItem()` and `Category::changeItem()` calls. Information about changes to Category attributes should be communicated by the subclass via `AttrBundle::setAttr()`.

Category also supports methods `Category::beginBlockChanges()` and `Category::endBlockChanges()` that can be called by subclasses to indicate the start and end of a block of notifications. `Category::replaceItemList()` can be called by subclasses when it wants to replace the current list of Item(s) by a new list. The Category base class computes any changes between its previous list and the new "list", updates its list and notifies interested listeners of any changes. None of `Category::addItem()`, `Category::changeItem()`, `Category::removeItem()` or `Category::replaceItemList()` should be called prior to the call to `Category::startMonitor()`.

P5

A subclass typically makes zero or more `Category::addItem()` and `Category::setAttr()` calls, followed by a `Category::endExists()` call followed by zero or more `Category::addItem()`, `Category::changeItem()`, `Category::removeItem()` and `Category::setAttr()` calls.

A Category subclass can also inform interested listeners of application-specific error notifications using `Category::notifyError()`. Error notifications are passed to **CategoryErrorListener** instances that are registered via `Category.addErrorListener()`.

The anticipated use of information in Category is by the client side code. Thus, the Java implementation of the CategoryListener interface is covered in detail in the Obtaining information about Category(s) and Item(s) on the client-side section. The information in that section can be applied to server-side components requiring information from a Category via the C++ CategoryListener API.

Plug-in a Category into the Rhino Infrastructure

CategoryFactory is the factory class for Category objects. CategoryFactory methods are mostly used by the **Category Service**, described in **sysadmd(1M)**, to fulfill requests from remote clients. They can also be used by any server-side components that require information from a Category. Category subclasses use the macros defined by CategoryFactory.h.

The steps required to make information about a Category of selector *catName* available to the rest of the system are detailed below. The Category instance will hold information about Item(s) of type *catName*

1. Implement a subclass of Category called *catName*.
    1. The subclass must have a void constructor. Typically, this calls the Category base class constructor with the argument *catName*.
    2. Use the convenience macro SaCATEGORY_REF_DECL, provided by CategoryFactory.h, in the header file to provide declaration for the routines used by the Category Service for obtaining Category instances.
    3. In most cases, only one instance of a particular subclass of Category should exist in an address space. To enforce this, subclasses should protect their constructors and use the convenience macro SaCATEGORY_FRIEND_DEF, provided by CategoryFactory.h, in the class declaration in the header file. This allows CategoryFactory access to the protected constructors.
    4. Use the convenience macro SaCATEGORY_REF_DEF, provided by CategoryFactory.h, in the c++ file to provide the definition for the routines used by the Category Service for obtaining Category instances. This in turn will use the void constructor.
2. Create a library called *catName*.so.
3. Install it in /usr/sysadm/category/

The above steps will allow clients to obtain the Category instance for *catName*. To avoid Category name clashes, applications should attach a product specific prefix to their categories. For example, FailSafe Manager and Miser Manager can use a category by name *ResourceCategory*, to refer to different entities. To avoid name clashes, the two categories could be named *fsmgrResourceCategory* and *msmgrResourceCategory*.

Consider, for example, the *UserAccountCategory* Category. In order to plug-in this Category into the Rhino infrastructure, create a library with the naming convention UserAccountCategory.so, with the entry points described above and install it in /usr/sysadm/category. The Category Service responds to client requests for a Category with selector *UserAccountCategory* by interfacing with UserAccountCategory.so to obtain information about the user account Item(s) and passes this information to the clients.

Code Snippet

Consider a Category named *rhexampRhinoExampleCategory* which is a collection of Item(s), one Item for each file in a particular directory. The Attributes of each Item correspond to the file name, file permissions and contents of the file. The implementation of this Category uses the fam(1M) AF obtaining the existing Item(s) in the system and monitoring of future changes is given below.

Header File: rhexampRhinoExampleCategory.h

```
#pragma once

#include <sys/types.h>

#include <sysadm/fam.h>

#include <sysadm/Category.h>
#include <sysadm/CategoryFactory.h>

namespace rhexamp {

using namespace sysadm;

SaCATEGORY_REF_DECL(rhexampRhinoExampleCategory);

//
// rhexampRhinoExampleCategory maintains an Item for each known
// RhinoExample.
//
class rhexampRhinoExampleCategory : public Category {

  protected:

    rhexampRhinoExampleCategory();
    virtual ~rhexampRhinoExampleCategory();

    // Start monitoring the system.
    virtual void startMonitor();

    // Allow CategoryFactory to create us.
    SaCATEGORY_FRIEND_DEF(rhexampRhinoExampleCategory);

  private:

    // Intentionally undefined.
    rhexampRhinoExampleCategory(const rhexampRhinoExampleCategory&);
    rhexampRhinoExampleCategory& operator=(const rhexampRhinoExampleCategory&);

    Item createItem(const char* exampleName);
    void processFamEvent(FAMEvent& event);

    static void famInput(void* clientData, int id, int fd);
```

96

COPY

```
            }
        }
        break;
    case FAMChanged:
        {
            Item item(createItem(event.filename));
            if (item.getSelector() != "") {
                changeItem(item);
            }
        }
        break;
    case FAMDeleted:
        removeItem(event.filename);
        break;
    case FAMEndExist:
        endExists();
        break;
    }
}

//
//  void rhexampRhinoExampleCategory::famInput(void* clientData, int, int)
//
//  Description:
//      Input callback that gets called when we get a FAM event.
//
//  Parameters:
//      clientData  ClusterCategory* (this is a static method).
//
void rhexampRhinoExampleCategory::famInput(void* clientData, int, int)
{
    rhexampRhinoExampleCategory* self = (rhexampRhinoExampleCategory*)clientData;

    FAMEvent event;
    while (FAMPending(&self->_famConn) == 1) {
        if (FAMNextEvent(&self->_famConn, &event) != -1) {
            self->processFamEvent(event);
        }
    }
}

//
//  void rhexampRhinoExampleCategory::startMonitor()
//
//  Description:
//      Set up our FAM connection.
//
//  Returns:
//      0 if successful, -1 if error.
//
void rhexampRhinoExampleCategory::startMonitor()
{
    if (FAMOpen(&_famConn) == 0) {
        _famStarted = true;
        FAMMonitorDirectory(&_famConn, RHINO_EXAMPLE_DIR,
                            &_configDir, NULL);
        _inputId = AppContext::getAppContext().registerMonitor(
            FAMCONNECTION_GETFD(&_famConn), famInput, this);
    } else {
        endExists();
    }
```

```
}

} // namespace rhexamp
```

rhexampRhinoExampleCategory.c++ is compiled into a library called rhexampRhinoExampleCategory.so and installed in /usr/sysadm/category. This makes information about rhexampRhinoExampleCategory and its Item(s) available to the rest of the Rhino infrastructure.

Obtaining information about Category(s) and Item(s) on the client-side: Java

All application-specific entities are instances of Item. Further, all application-specific categorie instances of Category. No subclassing is required by the developer of specific application. The steps in obtaining information about Category(s) and Item(s) are:

1. Obtaining a handle to a Category instance.
2. Obtaining information about Item instances.

Obtaining a handle to a Category instance

Category instances are obtained via a HostContext object. When writing Task UI interface, a HostContext object will be available for you from the Task infrastructure. The same applies to writing an ItemView, etc.

Internally, the HostContext object is obtained when a user successfully logs in to a server machine.

If the HostContext object is *hostContext*, then a client can obtain a handle to "UserAccountCategory" by using the following code:

```
Category cat = hostContext.getCategory("UserAccountCategory");
```

This is an asynchronous call that returns an handle to the Category before it receives a response f[illegible] the server. The client can use this handle to add CategoryListener instances for obtaining informati[illegible] error is encountered in communication with the server or loading the specific Category instance requested, this is handled as a fatal connection error by the infrastructure and the client will exit aft[illegible] error message is acknowledged by the user.

Obtaining information about Items

Clients interested in information about a Category instance can create a subclass of CategoryListener and register for notifications by passing a CategoryListener instance to Category.addCategoryListener(). The specific Items of interest are indicated by the NotificationFilter parameter. The NotificationFilter also specifies whether the CategoryListener instance is interested in notifications about the Category attributes. Call Category.removeCategoryListener() to unregister interest in notification.

Category base class notifies registered CategoryListener instances about Item(s) discovered (by

P7

COPY

```
    int _inputId;
    FAMConnection _famConn;
    FAMRequest _configDir;
    bool _famStarted;
};

} // namespace rhexamp
```

C++ File: rhexampRhinoExampleCategory.c++

```
#include <sys/stat.h>
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

#include <sysadm/format.h>
#include <sysadm/AppContext.h>
#include <sysadm/Log.h>

#include <rhexamp/RhinoExample.h>
#include "rhexampRhinoExampleCategory.h"

namespace rhexamp {

SaCATEGORY_REF_DEF(rhexampRhinoExampleCategory);

//
// Constructor.
//
rhexampRhinoExampleCategory::rhexampRhinoExampleCategory()
: Category("rhexampRhinoExampleCategory")
{
}

//
// Destructor.
//
rhexampRhinoExampleCategory::~rhexampRhinoExampleCategory()
{
    if (_famStarted) {
        FAMClose(&_famConn);
        AppContext::getAppContext().unregisterMonitor(_inputId);
    }
}

//
//  Item rhexampRhinoExampleCategory::createItem(const char *exampleName)
//
//  Description:
//      Respond to a FAM event that indicated that a new example has
//      been created.  This also gets called at startup for each
//      example that already exited.
//
//      Make sure example is valid, and if so put together Example
//      attributes.
//
//  Parameters:
//      exampleName  Name of the example that showed up.
//
//  Returns:
//      Newly created item corresponding to "exampleName".
//
Item rhexampRhinoExampleCategory::createItem(const char *exampleName)
{
    char exampleFile[PATH_MAX];
    (void)SaStringFormat(exampleFile, sizeof exampleFile,
                         "%s/%s", RHINO_EXAMPLE_DIR, exampleName);

    struct stat f;
    if (stat(exampleFile, &f) == -1) {
        return Item("", "");
    }

    FILE *fp = fopen(exampleFile, "r");
    if (fp == NULL) {
        return Item("", "");
    }

    char buf[100];
    char *type = fgets(buf, sizeof buf, fp);
    (void)fclose(fp);

    if (type == NULL) {
        return Item("", "");
    }

    char *pc = strchr(buf, '\n');
    if (pc) {
        *pc = '\0';
    }

    Item item(getSelector(), exampleName);
    item.setAttr(Attribute(RHINO_EXAMPLE_NAME, exampleName));
    item.setAttr(Attribute(RHINO_EXAMPLE_TYPE, type));
    item.setAttr(Attribute(RHINO_EXAMPLE_MODE, (long long)f.st_mode));

    return item;
}

//
//  void rhexampRhinoExampleCategory::processFamEvent(FAMEvent &event)
//
//  Description:
//      Process a single FAM event.
//
//  Parameters:
//      event  The event to process.
//
void rhexampRhinoExampleCategory::processFamEvent(FAMEvent &event)
{
    Log::trace(getSelector(), "Got a fam event");
    switch (event.code) {
    case FAMExists:
    case FAMCreated:
        {
            Item item(createItem(event.filename));
            if (item.getSelector() != "") {
                addItem(item);
```

P8

COPY

subclasses) in the system or Item(s) that are later added via `CategoryListener.itemAdded()` calls, Item changes via `CategoryListener.itemChanged()` calls, and Item removal via `CategoryListener.itemRemoved()`. Notifications about Category Attribute(s) discovered (by subclasses) in the system or Attribute(s) that are later added are via `AttrListener.attrAdded()` calls, Attribute changes via `AttrListener.attrChanged()` calls, and Attribute removal via `AttrListener.attrRemoved()`. When `Category.addCategoryListener()` is called, Category sends the listener its current list of Item(s) and Attribute(s) via `CategoryListener.itemAdded()` and `AttrListener.attrAdded()` calls. End of notification of the current state is signaled by a `CategoryListener.endExists()` call, if the Category itself has received this notification from its subclasses. Else, `CategoryListener.endExists()` will be called when Category receives this notification from its subclasses.

A CategoryListener can expect to receive zero or more `itemAdded()` and `attrAdded()` calls, followed by an `endExists()` call followed by zero or more `addItem()`, `changeItem()`, `removeItem()`, `attrAdded()`, `attrChanged()` and `attrRemoved()` calls. The `endExists()` call signals that the Category has communicated the entire set of Item(s) discovered in the system to the CategoryListener.

Category base class passes `Category.beginBlockChanges()` and `Category.endBlockChanges()` notifications to identically named methods on registered CategoryListener instances.

The following code illustrates how information can be obtained about Category(s) and Item(s) on the client-side. A CategoryAdapter is a default implementation of CategoryListener. The code below can be used in the Create A User Account Task to verify that the user account name does not already exist on the server.

```
Category cat = hostContext.getCategory("UserAccountCategory");
cat.addCategoryListener(
    new CategoryAdapter() {
        public void itemAdded(Item item) {
             if (item.getString(NAME).equals(userInputName)) {
                 // UserAccount with name userInputName already exists
                 // Steps to signal error ...
             }
        }
        public void endExists() {
            // UserAccount with name userInputName does not exist
            // Steps to signal successful verification ...
        }
    }, NotificationFilter.ALL_ITEMS);
```

The above method can be used when the user account name is not the same as the selector of the Item. If the client has the selector (unique name) of the Item, then there are two other ways of determining if the user account that was specified already exists. The following code uses a CategoryListener with a NotificationFilter that expresses interest in only one Item with a specified selector. If an Item with that selector exists, Category will pass the Item state to the CategoryListener via `CategoryListener.itemAdded()` followed by an `CategoryListener.endExists()`. If an Item with that selector does not exist it will send an `CategoryListener.endExists()` notification.

```
 NotificationFilter filter = new NotificationFilter();
 filter.monitorItem(selector);

 cat.addCategoryListener(
    new CategoryAdapter() {
        public void itemAdded(Item item) {
            // UserAccount with unique name "selector" already exists
            // Steps to signal error ...

        }
        public void endExists() {
            // UserAccount with  unique name "selector" does not exist
            // Steps to signal successful verification ...
        }
    }, filter);
```

Another way of obtaining an Item if the client has the selector is to use the `Category.getItem(` passing a **ResultListener** to `Category.getItem()`. `Category.getItem()` calls the `succeeded` of the ResultListener if an Item with the specified selector exists in the system. Use the `getResul` method of **ResultEvent** to get the Item. The Object returned by `getResult()` should be cast to an Item. `getItem()` calls the `failed()` method of the ResultListener if an Item with the specified selector does not exist in the system.

```
  cat.getItem(selector, new ResultListener() {
      public void succeeded(ResultEvent event) {
          // UserAccount with unique name "selector" already exists
          // Steps to signal error ...
      }

      public void failed(ResultEvent event) {
          // UserAccount with  unique name "selector" does not exist
          // Steps to signal successful verification ...
      }
  });
```

Similar to Category.getItem(), `Category.getItemCount()` can be used to get the number of Items in a Category and `Category.getItemList()` can be used to get the list of Items in a Category.

Putty-8047

Related: Item & Category

Category Names on the Client and Server

When creating certain Rhino UI components such as ItemTables, ItemViews, TreeViews, and ResultViews it is necessary to tell the component which Category it will be displaying. The UI components use this information to lookup resources that describe the Category, and to contact the server to request that the server send the Items in the Category. Simply passing the selector of the Category to the these UI components is not sufficient because the selector of the Category doesn't contain enough information to allow the components to find resources associated with the Category. To solve this problem, the name of a Java package that contains the resources relating to the Category should prepended to the Category's selector. The name of the resource file should be the same as the selector of the Category, but with a "P" appended.

For example, the resource file that controls the RhinoExampleCategory is named `.../com/sgi/rhexamp/category/rhexampRhinoExampleCategoryP.properties`. To request that an ItemTable show the RhinoExampleCategory, you would pass `com.sgi.rhexamp.category.rhexampRhinoExampleCategory` as the name of the Category.

This long name of the Category is referred to as the "fully qualified Category name" or the "package qualified Category name" in the API documents to distinguish it from the Category selector, which is referred to as "category name" or "category selector".

Since the package qualified Category name contains the Category selector, the methods that take a package qualified Category name can easily derive the Category selector when it is needed. The infrastructure always uses the category selector when communicating with the server.

In cases where the "package qualified Category name" is requested, it is legal to pass the Category selector instead. The UI components will be able to contact the server and request information about the appropriate Category, but it will have to use default display methods, since they don't have any information about how to display the Items.

Putty - 8048

Rhino UI Principles

XXX This is incomplete. It's taken from Wes' "UI Design Problems and Solutions" brown-bag slides.
XXX The layout sucks.

When designing a user interface with Rhino, try to follow these guidelines. Each is based on XXX

Problem: **Users don't have all the information they need to complete a task.**

Solution: **Make prerequisites explicit and complete.** Before letting users proceed in a multi-page Task, tell them what information they're going to need to provide.

Problem: **Users experience information overload.**

Solution: **Hide unnecessary details.**

Problem: **Users lack confidence in the GUI.**

Solution: **Always show accurate and current state.**

Solution: **Tell user what will and did happen.** This is one of the uses of the ResultView.

Problem: **Users' time is precious.**

Solution: **Identify problems as soon as possible.**

Solution: **Don't make the user type something the system knows.**

Solution: **Preserve data the user enters.** One way Rhino supports this is through TaskData, which is shared by both Form and Guide interfaces to a Task. If the user enters a value on one page of a Guide, and then decides they'd rather switch to the Form, the value they entered is still displayed. Similarly, if they enter a value on the first page of a Guide, proceed to the next page, and then return to the first page again, they don't have to

Problem: **Users don't know what to do next.**

Solution: **Let the user know what can be done next.** When a Task is successfully completed, show the user the list of Tasks which they're most likely to want to perform next. Rhino provides the ResultView for this.

P11

OOSTEO" E9S66TOS

Rhitty-8049

How to Customize the Task Manager

The Task Manager is a front end to all of the Tasks in a Rhino-based product. The Task Manager is customized for each Rhino product, but the basic interface appears the same for each product so that Users are presented with a common look and feel. Basic customization is accomplished through the creation of a product-specific properties file. It is also possible to plugin Java classes to handle more complex situations.

The Task Manager window has four parts, each of which can be customized to some degree.

- **Frame Title**
  By default, the Task Manager Frame Title will display a static, customizable string that includes the name of the server. The Frame Title can also be customized to display arbitrary dynamic strings such as server state information.
- **Table of Contents**
  On the left-hand side of the Task Manager window is the *Table of Contents* panel. The Table of Contents displays a set of links to product-specific pages. For example, a typical Table of Contents contains links to an Overview page, Search page, and a set of pages containing logically related Tasks.
- **Display Area**
  When the User chooses one of the page links in the Table of Contents, the corresponding page is displayed on the right-hand side of the Task Manager window called the *Display Area*. The Display Area can display three types of pages:

  1. **Text Page**
     A *Text Page* contains informational text. For example, the Overview page is typically a text description of how the Task Manager for the product works and describes the other pages available.
  2. **Task List Page**
     A *Task List Page* contains links to logically-related Tasks and Tasksets. When any of these links is activated, the corresponding Task or Taskset is launched in a separate window.
  3. **Class Page**
     A *Class Page* is a page that is implemented as a Java class and plugged in by the developer of a specific product. For example, Rhino provides a Search page plugin.
- **Button Bar**
  At the bottom of the Task Manager window is a button bar. A *Close* button is provided by default and will always appear as the rightmost button. Product-specific buttons may be added to the button bar.

For the duration of this document, let us assume that you are customizing the Task Manager for the product: *Rhino Example*.

The Task Manager Properties File

Basic customization of Task Manager is accomplished through the product-specific properties file called *TaskManagerP.properties*. This file typically resides in the top of your package hierarchy. For example,

```
myWorkArea/package/com/sgi/rhexamp/TaskManagerP.properties
```

The property names are documented in the Rhino class TaskManagerProperties and default values, when provided, exist in com.sgi.sysadm.manager.TaskManagerP.properties.

Customizing the Table of Contents

First you will want to customize the header, or title shown in the Table of Contents panel. This is accomplished by defining a property in TaskManagerP.properties as follows:

```
TaskManager.TOC.title = <B>Rhino Example Manager</B>
```

Next, you will want to create the product-specific list of page links. This is accomplished via an ordered property set called TaskManager.TOC.item<n>, where each item represents one page link or a separator. For each page, you specify a page type, a page title, and a page target.

For example, to specify a text overview page as the first page, you would add the following properties to the TaskManagerP.properties file:

```
TaskManager.TOC.item0 = text
TaskManager.TOC.item0.title = Overview
TaskManager.TOC.item0.target = \
    <P> Rhino Example Manager Graphical User Interface \
    provides access to the tasks that help you set up and \
    administer your Rhino Example objects. \
    <P>The tasks are organized into the categories \
    described below.  To view a category, click on it in the \
    <P> RhinoExample Task Manager Graphical User Interface (GUI) \
    provides access to the tasks that demonstrate use of \
    the Rhino infrastructure. \
    <P> \
    The tasks are organized into the categories described below. \
    To view a category, click on it in the column at left.<P>\
    <B>Overview</B> -- \
    Display this overview document. <P> \
    <B>Search</B> -- \
    Use keywords to search for a specific task. <P>\
    <B<RhinoExample Tasks</B> -- \
    Example tasks that demonstrate the use of the Rhino infrastructure.
```

Note that for text pages, the 'target' contains the actual text to be displayed on the page.

Now let's imagine that you want the next page to be a search page that allows Users to find the Task they are interested in via keyword. Rhino provides a Class Page plugin SearchPanel that indexes all of the Tasks and Tasksets by keyword. Here's how the page would be specified in TaskManagerP.properties:

```
TaskManager.TOC.item1 = class
TaskManager.TOC.item1.title = Search
```

COPY

P12

```
TaskManager.TOC.item1.target = \
     com.sgi.sysadm.manager.taskManager.SearchPanel
```

The target for a page of type *class* is the CLASSPATH relative name of the page plugin, which must implement the TaskManagerPanel interface.

Next we'll add a separator in the Table of Contents, which does not require the title or target specifiers:

```
TaskManager.TOC.item2 = separator
```

Finally, we'll add a Task List page. Task List pages display an optional list of Tasksets (also known as Metatasks), a separator, and then an optional list of Task Groups. Tasksets provide guidance in accomplishing a high level goal that may involve multiple tasks. Task Groups are (possibly ordered) lists of Tasks that are closely related, usually by the type of object they operate on. For example, Tasks that all operate on User accounts are likely to be in the same Task Group.

```
TaskManager.TOC.item3 = tasklist
TaskManager.TOC.item3.title = RhinoExample Tasks
TaskManager.TOC.item3.target = RhinoExampleTasks
```

The target for a Task List page will be used as a key to optional property sets that describe the list of Tasksets (also known as Metatasks) and Task groups to display on the page. For example, the RhinoExample Tasklist page will have a single Task group:

```
TaskManager.RhinoExampleTasks.taskGroup0 = MyTaskGroup
TaskManager.RhinoExampleTaskGroup.introText = \
     <B>Rhino Example Tasks</B>
```

The introText property is the text to display at the top of the Task Group. The Task Group itself is installed in the TaskRegistry on the server, in a directory named "MyTaskGroup". See Plugging in a Task Group later in this document for details.

If we also wanted to display a list of Tasksets on the RhinoExample Tasks page, the properties might look like this:

```
TaskManager.RhinoExampleTasks.metatasksText = The following tasksets can \
     help you keep your system  up and running in production \
     mode.  Find the taskset that suits \
     your needs, then click to launch it.

TaskManager.RhinoExampleTasks.metataskItem0 = \
     com.sgi.rhexamp.metatask.FirstExampleTaskset
TaskManager.RhinoExampleTasks.metataskItem1 = \
     com.sgi.rhexamp.metatask.SecondExampleTaskset
```

The metatasksText is displayed above the entire list of Taskset links (and is optional).

Each metataskItem refers to another Properties file that describes the contents of the Taskset, which will launch in a separate window when activated. Here is an example of the contents of a Taskset properties file:

```
#
# Properties for First Example Taskset
#

Metatask.name = First Example Taskset
Metatask.keywords = keywords to help users find this Taskset

Metatask.text = \
     <B>Achieve a High Level Goal</B><P> \
     This taskset lists different ways to achieve a goal. \
     Here are some of the options: \
     <UL>\
     <LI><A href=task.com.sgi.fsmgr.task.ModifyClusterTask>
     <B>Modify an Example</B></A> -- \
     Set Example attributes \
     <LI><A href=task.com.sgi.fsmgr.task.DefineMachineTask>
     <B>Define an Example</B></A> \
     -- Create an Example. \
     </UL>
```

A Taskset (or Metatask) has three attributes set in its properties file: the name, keywords, and text. The text specified will be used to create a RichTextComponent that can contain links that launch Tasks, other Tasksets, or glossary entries when activated.

The font and color of the Table of Contents panel, title label, and page links can all be customized as well. See TaskManagerProperties for details.

Plugging in a Task Group

A Task List page can contain one or more Task Groups. Each Task Group is a (possibly ordered) list of Tasks that are closely related, usually by the type of object they operate on. For example, Tasks that all operate on User accounts are likely to be in the same Task Group.

Task Groups are specified in the properties for a Task List page rather than the Tasks themselves to allow new Tasks to be plugged in to the Task Manager without requiring the images to be re-shipped. For example, a new product could be created that adds Tasks to existing Task Groups, and they will automatically appear in the Task Manager the next time it is launched.

Tasks are plugged into Task Groups via the Task Registry on the server. Here are the steps needed to create a new Task Group named myTaskGroup in the Task Registry and add an ordered set of Tasks to that group:

1. cd myWorkarea/taskRegistry
2. mkdir MyTaskGroup
3. copy the Makefile from any existing Task Group or Task Category into MyTaskGroup (you should have an existing Task Group and Task Category if you created your Rhino ism using the Make Rhino Ism Task).
4. cd MyTaskGroup
5. For each Task you want to add:
    1. touch {4-digit order key}.com.sgi.{myProduct}.task.{taskName}
    2. p_modify -f {new file name from previous step}
6. p_modify Makefile
7. Edit the Makefile, replacing the existing file names with the new files added in the previous steps.
8. cd ..
9. p_modify Makefile
10. add MyTaskGroup to the list of directories to build

P13

11. cd myWorkarea/build
12. update the idb with the new Tasks
13. p_integrate
14. p_finalize

For example, if you wanted to add three Tasks to the MyTaskGroup named "AddRhinoExampleTask", "ModifyRhinoExampleTask" and "DeleteRhinoExampleTask", in that order, you would do the following:

- cd myWorkarea/taskRegistry
- mkdir myTaskGroup
- cp {otherTaskGroup}/Makefile MyTaskGroup
- cd MyTaskGroup
- touch 1000.com.sgi.myProduct.task.AddRhinoExampleTask
- touch 2000.com.sgi.myProduct.task.ModifyRhinoExampleTask
- touch 3000.com.sgi.myProduct.task.DeleteRhinoExampleTask
- update the Makefiles and the idb file

You will need to build and install the taskRegistry onto your server.

Adding Buttons to the Button Bar

By default the Button Bar at the bottom of the TaskManager window has a Close button. When pressed, the Close button will terminate the Task Manager application but any other windows launched from Task Manager will stay open. The Close button will always appear as the rightmost button.

Buttons are added by creating an ordered property set in the Task Manager properties file. For example:

```
TaskManager.buttonItem0 = First Button
TaskManager.buttonItem0.target = com.sgi.myProduct.myFirstPlugin
TaskManager.buttonItem1 = Second Button
TaskManager.buttonItem1.target = com.sgi.myProduct.mySecondPlugin
```

Each button is given a name that will be displayed on that button as well as a target class that should be launched when the button is activated. The target class must implement one of TaskManagerFrame or TaskManagerAction. The first button added will be the leftmost button on the button bar. Subsequent buttons will be added to the right of the previous button but always to the left of the Close button.

Customizing the Task Manager Frame Title

By default, the Task Manager Frame Title will display a static string that includes the server name. This static string may be customized via a property. For example:

```
TaskManager.frameTitle = RhinoExample Manager (on {0})
```

Where the argument {0} is replaced with the server name.

If you wish to have a dynamic title that, for example, changes when the state of an object on the server changes, then you will want to use a TaskManagerTitleRenderer. A title renderer is a class that is responsible for keeping the title string up to date. It can monitor the server and make updates as desired. To plugin a title renderer, you use a property in the Task Manager properties file. For example:

```
TaskManager.titleRenderer = com.sgi.myProduct.plugin.MyTitleRenderer
```

Adding Code that Runs at Startup

Some products need to run initialization code when their Task Manager starts up. For example, a product might want to set up default values for TaskData that will be used by product-specific tasks. A TaskManagerInitPlugin is where that default-setting code should reside. Multiple TaskManagerInitPlugins can be plugged in via the Task Manager properties file. For example:

```
TaskManager.initPlugins0 = com.sgi.myProduct.plugin.MyFirstInitPlugin
TaskManager.initPlugins1 = com.sgi.myProduct.plugin.MySecondInitPlugin
```

The initialization plugins will be run, in order, as the first operation when the Task Manager is launched.

Running Task Manager

Let's suppose you have created your TaskManagerP.properties file in myWorkArea/package/com/sgi/myProduct and CLASSPATH includes "myWorkArea/package". To launch your customized Task Manager, you would enter the following command:

```
java com.sgi.sysadm.manager.TaskManager -p com.sgi.myProduct
```

To launch the Task Manager programmatically, you need to know the CLASSPATH relative name of the product (so that Task Manager can find the product-specific properties file. For example:

```
void launchTaskManager() {
    // Go busy while launching
    _uic.busy(new ResultListener() {
        public void succeeded(ResultEvent event) {
            TaskManager tMgr = new TaskManager("com.sgi.myProduct");
            tMgr.initApp();
            tMgr.run(_hc, new RApp.RAppLaunchListener() {
                public void launchSucceeded(RApp.RAppLaunchEvent event) {
                    _uic.notBusy();
                }
                public void launchFailed(RApp.RAppLaunchEvent event) {
                    _uic.notBusy();
                }
                public void launchAlreadyRunning(
                    RApp.RAppLaunchEvent event) {
                    _uic.notBusy();
                }
            });
        }
        public void failed(ResultEvent event) {
        }
    });
}
```

See RApp for more information on launching a Rhino application.

P14

OOSTEO" E9S6STO9

Putty-8050

How to Write a Task

Outlined below are the basic steps involved in writing a Task for Rhino. Unless otherwise noted, the code examples below are for a Define User Account Task.

Before proceeding, you should familiarize yourself with Basic Concepts and at least look over the GUI Components, Architecture, and Task Internals documents.

1. Create the Task properties file

The Task properties file is required to exist in the same directory as the Task class. The name of the properties file must be the Task class name followed by "P.properties". For example, a Task subclass named "DefineUserAccountTask" must have a properties file named "DefineUserAccountTaskP.properties".

The Task properties file contains static information about the Task, including the Task title, privilege list, and whether or not the Task accepts operands. The Task properties file also contains User-visible labels and messages, and interface characteristics such as fonts, colors, and sizes.

Below is a sample of what the DefineUserAccountTaskP.properties file might contain. Letters have been used to identify the lines to distinguish them from the Java code examples that follow.

```
A: #
B: # Properties file for the Define User Account Task
C: #
D: Task.shortName = Define User Account
E: Task.longName = Define a new User Account
F: Task.keywords = define new add user account login home directory shell
G:
H: [...]
I:
J: #
K: # DO NOT LOCALIZE BELOW THIS LINE
L: #
M: Task.privList0 = addUser
N: Task.privList1 = listUsers
O: Task.acceptsOperands = false
P: Task.publicData0 = userName
Q: Task.ProductAttributes0 = com.sgi.psa
```

2. Implement the Task subclass

The Task subclass is the main entry point to the Task. Its functions are to verify prerequisites, initialize TaskData, coordinate the Task interface(s), and perform the Task operation when the User presses the OK button.

- **Constructor**
  The Task subclass constructor is responsible for initializing all public and private TaskData and adding TaskDataVerifiers for each TaskData attribute that needs to be verified when the User presses OK.

```
private TaskContext _taskContext;
private TaskData _taskData;
private ResourceStack _rs;
private HostContext _hc;
private Category _userCategory;

private static final String USER_NAME = "userName";

public DefineUserAccountTask(TaskContext taskContext) {
    super(taskContext);

    _taskContext = taskContext;
    _taskData = _taskContext.getTaskData();
    _rs = _taskContext.getResourceStack();
    _hc = _taskContext.getHostContext();

    _taskData.setString(USER_NAME, "");
    _taskContext.appendTaskDataVerifier(USER_NAME, new
                                        TaskDataVerifier() {
        public void dataOK(int browseFlag, Object context,
                           ResultListener listener) {
            verifyUserName(browseFlag, context, listener);
        }
    });

    [...]
}

public void verifyUserName(int browseFlag, Object context,
                           ResultListener listener) {
    String userName = _taskData.getString(USER_NAME);
    ResultEvent result = new ResultEvent(this);

    if (userName.length() == 0) {
        if (browseFlag) {
            listener.succeeded(result);
        } else {
            result.setReason(
                _rs.getString("Error.missingUserName"));
            listener.failed(result);
        }
    }

    // Check user name for syntactic problems (length,
    // unprintable characters, etc.).
    [...]

    // Check for existing user name
    verifyUniqueName(userName, listener);
}

public void verifyUniqueName(final String userName,
                             final ResultListener listener) {
    _userCategory = _hc.getCategory("UserAccountCategory");
    _userCategory.getItem(userName, new ResultListener() {
```

COPY

P15

```
56:            public void succeeded(ResultEvent event) {
57:                event.setReason(MessageFormat.format(
58:                    _rs.getString("Error.userExists"),
59:                    new Object[] { userName } ));
60:                listener.failed(event);
61:            }
62:
63:            public void failed(ResultEvent event) {
64:                listener.succeeded(event);
65:            }
66:        });
67:    }
```

- **Task.registerInterfaces()**
  Task.registerInterfaces() is abstract, therefore it must be implemented by the subclass. Its responsibilities are to:

  1. Create the Task interface classes (Form and/or Guide) and register those classes with the Task base class using Task.setForm() and Task.setGuide() respectively.
  2. Override the default title string if it is to include something other than *Task.shortName* and the server name (not shown in the example below).

```
68:    public void registerInterfaces() {
69:        setForm(new DefineUserAccountForm(_taskContext));
70:        setGuide(new DefineUserAccountGuide(_taskContext));
71:    }
```

- **Task.setOperands()**
  Task.setOperands() is only required if the property Task.acceptsOperands is set to true (the default). Note that setOperands() is synchronous, so it should not do any operand verification that involves a server request. Server verification should be deferred until verifyPrereqsBeforeCheckPrivs() or verifyPrereqsAfterCheckPrivs().

  Although the DefineUserAccountTask does not take operands, below is some sample code for the ModifyUserAccountTask, which accepts a single User Account as an operand.

```
72:    public void setOperands(vector Operands)
73:            throws TaskInitFailedException {
74:        if (operands == null || operands.size() == 0) {
75:            // The operands are optional
76:            return;
77:        }
78:
79:        if (operands.size() > 1) {
80:            throw new TaskInitFailedException(
81:                _rs.getString("Error.tooManyOperands"),
82:                TaskInitFailedException.INVALID_OPERANDS);
83:        }
84:
85:        // In the future, operands could be Items that are
86:        // dropped via the drag and drop interface.  In
87:        // Rhino 1.0, however, we can only pass String operands
88:        // at this time.
89:        if (!(operands.elementAt(0) instanceof String)) {
90:            throw new TaskInitFailedException(
91:                _rs.getString("Error.invalidOperandType"),
92:                        TaskInitFailedException.INVALID_OPERANDS);
93:        }
94:
95:        // Store the operand for later use
96:        _taskData.setString(USER_NAME, (String)operands.elementAt(0));
97:    }
```

- **Task.verifyPrereqsBeforeCheckPrivs()**
  Task.verifyPrereqsBeforeCheckPrivs() is provided as a hook for subclasses to do whatever verification is possible before privileges are obtained. The base class provides a trivial implementation of verifyPrereqsBeforeCheckPrivs() that always succeeds, so you do not need to implement this method if you have no prerequisites to verify or if all of your checking requires privileges.

  For illustrative purposes, imagine that the Define User Account Task allows the proposed U[illegible] name be set via setTaskDataAttr() (note that "userName" was declared as a public TaskData attribute in the Task properties file above). A prerequiste could be that the User name may not already exist on the server. Below is the code that would be used to verify the User name as a prerequisite. Note that the "userName" TaskDataVerifier, defined in lines 18-24 of the constructor, is being referenced here by name.

```
98:    public void verifyPrereqsBeforeCheckPrivs(ResultListener listener) {
99:        _taskContext.dataOK(USER_NAME, TaskDataVerifier.MAY_BE_EMPTY,
100:                           null, listener);
101:   }
```

- **Task.verifyPrereqsAfterCheckPrivs()**
  Task.verifyPrereqsAfterCheckPrivs() is provided as a hook for Task subclasses that need privileges in order to verify some of their prerequisites. The base class provides a trivial implementation of verifyPrereqsAfterCheckPrivs() that always succeds, so you do not need to implement this method if you have no privileged verification.

  Privileged verification requires a call to the version of Task.runPriv() that takes a ResultListener. A generic example follows.

```
102:   public void verifyPrereqsAfterCheckPrivs(ResultListener listener) {
103:       TaskData CLIArgs = new TaskData();
104:       CLIArgs.setAttr(_taskData.getAttr("prereqData"));
105:       [...]
106:
107:       OutputStream stream = runPriv("listUsers", CLIArgs, listener);
108:
109:       try {
110:           stream.close();
111:       } catch (IOException ex) {
112:           Log.debug("DefineUserAccountTask",
113:                     "unable to close listUsers stream");
114:       }
115:   }
```

- **Task.ok()**
  Task.ok() is responsible for sending a request to the server to perform the requested Task. In the most common case, a single privileged command is needed to perform the Task. When the Task is more complex, however, multiple privileged commands may be involved. The sample code below

covers the typical case. See the Task API documentation for details on other versions of Task.runPriv() that can be used in more complex Tasks.

```
    public void ok() {
        TaskData CLIArgs = new TaskData();
        CLIArgs.setAttr(_taskData.getAttr(USER_NAME));
        [...]

        OutputStream stream = runPriv("addUser", CLIArgs);

        try {
            stream.close();
        } catch (IOException ex) {
            Log.debug("DefineUserAccountTask",
                      "unable to close OutputStream from runPriv");
        }
    }
```

- **createResultViewPanel()**
  Task clients that want to display a ResultViewPanel after being notified that the Task has succeeded call Task getResultViewPanel(). That method checks to make sure the Task succeeded and then calls the abstract Task.createResultViewPanel(). The ResultViewPanel should show information about the object modified or created, if appropriate; text describing the result and consequences of the Task; and a list of Tasks that the User might logically want to invoke next.

```
    public ResultViewPanel createResultViewPanel() {
        return new ResultViewPanel(_taskContext, _rs,
                                   "UserAccountCategory",
                                   _rs.getString(
                                       "DefineUserAccountTask.succeeded"));
    }
```

3. Implement the Form subclass

The Form subclass sets up the visible components for the Form interface of a Task, and then binds the components to the appropriate TaskData attributes.

- **Constructor**
  The Form subclass constructor has no specific responsibilities. In practice, it is used to create aliases to the TaskContext and any other classes that will be shared with the Task subclass.

```
public class DefineUserAccountForm extends Form {

    private TaskContext _taskContext;
    private TaskData _taskData;

    public void DefineUserAccountForm(TaskContext taskContext) {
        super(taskContext);

        _taskContext = taskContext;
        _taskData = _taskContext.getTaskData();
    }
```

- **Form.createUI()**
  Form.createUI() is responsible for creating the visible components of the Form interface. It is not called until the Form is actually displayed. Form.createUI() should always call super.createUI() as its first act in order to create the Form icon and title at the top of the Form interface. See the API documentation for details.

```
    public void createUI() {
        super.createUI();

        ResourceStack rs = _taskContext.getResourceStack();

        FilteredTextField userName =
            new FilteredTextField(
                rs.getInt("DefineUserAccountForm.userNameFieldWidth"),
                FilteredTextField.BEEP);

        addTaskComponent(userName,
                         rs.getString(
                             "DefineUserAccountForm.userNameLabel"));
        StringJTextComponentBinder.bind(_taskData, USER_NAME, userName);

        // More visible components would be added here
        [...]
    }
```

4. Implement the Guide subclass

The Guide subclass is slightly more complicated than the Form interface because the developer breaks the interface into multiple pages, each of which may have its own TaskDataVerifier that gets called when the User presses the Next button to leave that page.

- **Constructor**
  The Guide subclass constructor has no specific responsibilities. In practice, it is used to create aliases to the TaskContext and any other classes that will be shared with the Task subclass.

```
public class DefineUserAccountGuide extends Guide {

    private TaskContext _taskContext;
    private TaskData _taskData;
    private ResourceStack _rs;

    public DefineUserAccountGuide(TaskContext taskContext) {
        super(taskContext);

        _taskContext = taskContext;
        _taskData = _taskContext.getTaskData();
        _rs = _taskContext.getResourceStack();
    }
```

- **Guide.registerPages()**
  Guide.registerPages() is responsible for creating and registering each of the GuidePages that make up the Guide. If your Guide has pages that only appear if the User selects certain options, those pages can be registered later using either Guide.appendPage() or Guide.insertPage(). See the API documentation for more details about creating GuidePages.

  The verification for a GuidePage is called when the User presses the Next button. The example below is very simple because there is only one input field on the page. TaskContext has additional

P17

versions of the dataOK() method that allow the developer to chain together a set of TaskDataVerifiers.

```
    public void registerPages() {
        GuidePage userNamePage =
            new GuidePage(_taskContext, "UserNamePage") {
                public void createUI() {
                    super.createUI();

                    FilteredTextField userName =
                        new FilteredTextField(_rs.getInt(
                            "DefineUserAccountForm.userNameFieldWidth"),
                        FilteredTextField.BEEP);

                    addTaskComponent(userName,
                        rs.getString("DefineUserAccountForm.userNameLabel")
                    StringJTextComponentBinder.bind(_taskData, USER_NAME,
                                                    userName);
                }
        }

        userNamePage.setVerifier(new TaskDataVerifier() {
            public void dataOK(final int browseFlag, final Object context,
                               final ResultListener linstener) {
                _taskContext.dataOK(USER_NAME, browseFlag, context,
                                    listener);
            }
        });

        appendPage(userNamePage);

        // Additional pages would be added here
        [...]
    }
```

Running a Task from the command line

```
% setenv CLASSPATH \
/usr/sysadm/java/swingall.jar:/usr/sysadm/java/sysadm.jar:(task workarea)
```

```
% java com.sgi.sysadm.manager.RunTask (package).(taskname) [operands]
```

See the RunTask documentation for a list of available runtime options.

Areas to be covered in a future revision

Integrating Tasks into Other Views

There are two ways that a set of Tasks can be associated with a particular view:

1. The list of Tasks is hardcoded in the view code or properties file.
2. The list of Tasks is retrieved from the TaskRegistry, filtered by ItemTester.

Sharing Code Among Multiple Tasks

1. **Subclassing**
2. **Libraries**

Tips on Asynchronous Programming

- **Why Asynchronous Calls are Needed**
- **Common Problems in Asynchronous Programming**

P18

Putty - 8051

How to use a Rhino IconRenderer

Table of Contents

Introduction

This document is a reference for SGI software engineers who will be using IconRenderers for Rhino applications. IconRenderer is a class that can generate an icon that represents a particular Item. Many of the Rhino infrastructure components use an IconRenderer to display the icon associated with an Item, including the ItemView, the ItemTable, the ResultView, and the TreeView. The IconRenderer gives the programmer the ability to define what icon an Item should use, and have that icon used everywhere that an Item's icon is displayed.

There is one IconRenderer for each Category. The IconRenderer is responsible for monitoring the Category and generating icons for any Items that listeners have expressed interest in.

For more information on IconRenderers in general, see the IconRenderer documentation API documentation.

Before you begin

Before you begin to create an IconRenderer for a particular Category, you need to understand the names and terms that the Rhino infrastructure uses in relation to Categories. See the The names of Categories on the client and on the server documentation for more information.

Specifying the IconRenderer to use for a Category

The HostContext keeps track of which IconRenderer to use for each Category. To specify a Category's IconRenderer, place the ICON_RENDERER resource in the Category's resource file. For example, to specify that the RhinoExampleCategory should use the "com.sgi.rhexamp.category.IconRenderer" class as its IconRenderer, the following entry would be made in

```
/com/sgi/rhexamp/category/rhexampRhinoExampleCategoryP.properties:

com.sgi.rhexamp.category.rhexampRhinoExampleCategory.iconRenderer = com.sgi.rhexamp.
```

If this resource is not specified, then the HostContext creates a ResourceBasedIconRenderer object for the Category.

Using the ResourceBasedIconRenderer

The ResourceBasedIconRenderer is a subclass of IconRenderer that is used to display icons if th ICON_RENDERER resource specified (see Specifying the IconRenderer to use for a Category). To ResourceBasedIconRenderer, you must place specific resources in the Category's resource file.

Using the same icon for all the Items in a Category

If you want to use the same icon for all the Items in a Category, then only the "icon" resource is needed, defined as *<name>.icon* (see the DEFAULT_ICON documentation for more information). The icon described by the "icon" resource can be either classpath-relative pathnames to .gif or .jpg icons, or they can be package-qualified names of FtrIcon. For example, if all of the Items in the RhinoExample Category should show an icon of a rhino, then the following lines would be placed in the resource file:

```
A: RHINO_EXAMPLE_CATEGORY=com.sgi.rhexamp.category.rhexampRhinoExampleCategory
B: $(RHINO_EXAMPLE_CATEGORY).icon = /com/sgi/sysadm/ui/images/sysadm.gif
```

Using different icons for each Item in a Category

To show a different icon for each Item in the Category, add the "iconBasedOn" resource, defined as *<name>.iconBasedOn*, where <name> is the package-qualified name of the Category (see the ICON_BASED_ON documentation for more information). This resource specifies which of the Item's Attributes the icon will be based on.

In conjunction with the "iconBasedOn" resource, there should be "icon" resources, defined as *<name>.icon.<Attribute's value>*, where <Attribute's value> is one of the values that the Attrib specified by "iconBasedOn" can have. (See the ICON documentation for more information). If no resource is found that matched the value of the Attribute specified by "iconBasedOn", then the defa icon (as described above) will be used.

For example, in the RhinoExample category, the icon is based on the type, so the following entries are made in the resource file:

```
A: RHINO_EXAMPLE_CATEGORY=com.sgi.rhexamp.category.rhexampRhinoExampleCategory
B:
C: $(RHINO_EXAMPLE_CATEGORY).iconBasedOn = type
D: $(RHINO_EXAMPLE_CATEGORY).icon = com.sgi.rhexamp.ftr.Unknown
E: $(RHINO_EXAMPLE_CATEGORY).icon.Clock = com.sgi.rhexamp.ftr.Clock
F: $(RHINO_EXAMPLE_CATEGORY).icon.Printer = com.sgi.rhexamp.ftr.Printer
G: $(RHINO_EXAMPLE_CATEGORY).icon.NetscapeExecutable = com.sgi.rhexamp.ftr.NetscapeE
```

P 19

COPY

With the resource defined as shown, then the icon displayed will be based on the "type" Attribute of the Item. For example, if the "type" is "Clock", then the FtrIcon `com.sgi.rhexamp.ftr.Clock` will be used. If the "type" Attribute is not one of "Clock", "Printer" or "NetscapeExecutable", then the `com.sgi.rhexamp.ftr.Unknown` will be displayed.

Another way to show different icons for each Item in the Category is to use the "iconModifiers" resource, defined as *<name>.iconModifiers* (see the ICON_MODIFIERS documentation for more information). This resource defines an array of Attributes of the Item that will be passed to the `set` method of FtrIcon. The FtrIcon can then use the Attributes to choose how to display the icon. The Attributes defined by the "iconModifiers" resource will be passed to the FtrIcon that is created. The Attributes will be passed to the FtrIcon both in the case where a default icon is used and in the case where a specific icon is used. These Attributes will be ignored if the icon is not an FtrIcon.

For example, if the FtrIcons for the RhinoExampleCategory could draw themselves differently based on the "mode" Attribute of the Item, then add the following to the resource file:

```
A: $(RHINO_EXAMPLE_CATEGORY).iconModifiers0 = mode
```

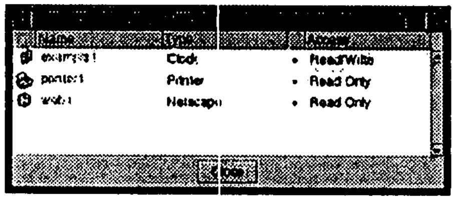
In this case, the Item's "mode" Attribute will be passed to whatever FtrIcon is used (which - as described above - depends on the "type" Attribute).

Using a subclass of IconRenderer

To provide complete control of the icon that is used for an Item, it is also possible to subclass the IconRenderer class and provide the necessary Java code for rendering icons. See the IconRenderer documentation for more information. It is also possible to subclass the ResourceBasedIconRenderer to add to the existing functionality of the ResourceBasedIconRenderer.

Putty- 8052

How to write a Rhino ItemTable



Name	Type	Access
example1	Clock	• Read/Write
printer1	Printer	• Read Only
web1	Netscape	• Read Only

Table of Contents

Introduction

This document is a reference for SGI software engineers who will be writing ItemTables for Rhino applications. An ItemTable in the Rhino Architecture is a UI Component that displays all the Items that exists in a particular Category (or Association). The ItemTable is not meant to display all of the information about each Item (that is the job of the ItemView). The ItemTable should show the information that the user is most likely to be interested in seeing, and is limited to displaying information that can fit into the cells of the table.

Overview of the ItemTable

The ItemTable is composed of a number of columns, each column representing one piece of information from the Item. Each Item is represented by a row of the table. Each column has a descriptive header. If the user clicks on the header, the table is sorted based on the associated column.

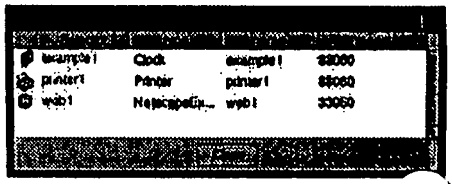
The first column of the ItemTable is reserved for the Icon that represents the Item. The Icon is not controlled by the ItemTable, but is generated by the ResourceBasedIconRenderer. See the tutorial on using ResourceBasedIconRenderer for details on how to control the Icon.

Before you begin

Before creating an ItemTable for a particular Category, it is necessary need to understand the names and terms that the Rhino infrastructure uses in relation to Categories. See the The names of Categories on the client and on the server documentation for more information.

Customizing the Columns of the ItemTable

No-Code ItemTables



example1	Clock	example1	[illegible]
printer1	Printer	printer1	[illegible]
web1	NetscapeEx...	web1	[illegible]

While in the early stages of writing Categories, it may be desirable to show an ItemTable that shows all of the Items of a Category. The ItemTable supports this idea by means of a "no-code" ItemTable. This version of an ItemTable is not designed for use in a shipping Rhino application, but can be of great assistance while investigating the Rhino Infrastructure or for giving preliminary demos. No code or resource files need to be written to us "no-code" ItemTable - it can be launched as soon the server side Categories have been written ano Rhino infrastructure has been installed on the client. An example of this "no-code" ItemTable is sh to the right. To launch a "no-code" ItemTable, follow the instructions in the section titled How to launch ItemTables. While this ItemTable shows a lot of information, it is not suitable for use in a shipping code. To turn this ItemTable into a shippable ItemTable, it is necessary to provide resources that describe the order in which the column will be displayed, suitable headers for the columns, and ways to internationalize the ItemTable. The rest of the document will describe how to accomplish this.

The *column* property

The manner in which the Items are displayed in the columns can be completely controlled by a resource file. The most important resource entries are the ones that name the columns that will be displayed. The

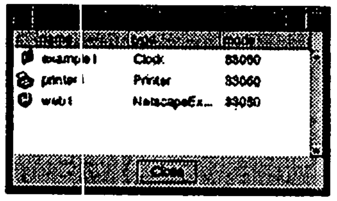
P 21

names of the resources follow the form *<Category name>.ItemTable.column<n>*, where <Category name> is the name of the Category and <n> represents integers starting at 0 that represent order in which the columns should be displayed. (see the COLUMNS documentation for more details). For example, the resource file that controls the RhinoExampleCategory could contain the following lines (the letters in the first column are for reference purposes only):

```
A: com.sgi.rhexamp.category.rhexampRhinoExampleCategory.ItemTable.column0=name
B: com.sgi.rhexamp.category.rhexampRhinoExampleCategory.ItemTable.column1=type
C: com.sgi.rhexamp.category.rhexampRhinoExampleCategory.ItemTable.column2=mode
```

Because the first part of each line is identical, it is common to use macros to shorten the lines of the resource file and to make the file easier to read. An example of the same resources using macros is shown below.

```
A: RHINO_EXAMPLE_CATEGORY=com.sgi.rhexamp.category.rhexampRhinoExampleCategory
B: ITprefix=${RHINO_EXAMPLE_CATEGORY}.ItemTable
C:
D: ${ITprefix}.column0=name
E: ${ITprefix}.column1=type
F: ${ITprefix}.column2=mode
```



example1	Clock	$3000
printer1	Printer	$3060
web1	NetscapeEx...	$3050

The three "column" lines (D - F) describe both the names of the columns and the order in which the columns will be displayed in the ItemTable. The names will be used later in the resource file to associate resources with particular columns. In this example, the names of the columns correspond exactly with the names of the Attributes in the Item that will be displayed in the column. By naming the columns in this manner, the ItemTable can use default behavior and automatically associate the correct Attribute with the column. It is also possible to give the columns names that are not the same as the names of Attributes. In that case, it may be necessary to use the "basedOn" property (defined below) to tell the ItemTable which Attribute is associated with a column.

Running an ItemTable with only the 5 lines described above in the resource file will result in an ItemTable that is shown on the right. Notice that the order of the columns (from left to right) is "name", "type", and then "node", which was as specified in the resource file. ItemTable has used the names of the columns as the labels for the columns. Information about how to customize the labels is described below. The ItemTable is using the default "toString" method (methods are described below). This is the simplest method, and uses the results of calling Java's `toString` method on the value of the Attribute.

The *basedOn* property

In the example resource file shown above, the names of the columns were defined to be the same as the Item's Attributes that they represented. This allowed the ItemTable to automatically show the value of the Attribute in the columns. It is sometimes desirable to use different names for the columns than the Attributes that control them. This can be to make the resource file more readable or because there may not be a one to one correspondence between the Attributes in the Item and the columns that are displayed.

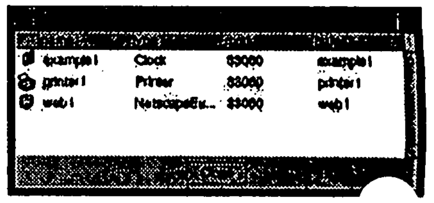
If a column is given a name that does not correspond to the name of an Attribute, the "basedOn" property is used to tell the ItemTable which Attribute the column represents. The "basedOn" resources are defined as: *<Category name>.ItemTable.basedOn.<column>*, where <Category name> is the name of the Category, and <column> is the name of a column. (See the BASED_ON documentation for more details).

The `stringRenderer`, `richTextRenderer`, and `componentRenderer` methods (as described below) do not require that the column be associated with a particular Attribute. When using these methods, it is not necessary to specify the "basedOn" property even if the name of the column does not correspo n Attribute. All the other methods, including the default "toString" method, require that the colu associated with a particular Attribute of the Item.

For example, suppose that for some reason we wish to display the name Attribute twice, once as the first column, and once as the last column. A resource file as follows would do just that:

```
A: RHINO_EXAMPLE_CATEGORY=com.sgi.rhexamp.category.rhexampRhinoExampleCategory
B: ITprefix=${RHINO_EXAMPLE_CATEGORY}.ItemTable
C:
D: ${ITprefix}.column0=name1
E: ${ITprefix}.column1=type
F: ${ITprefix}.column2=mode
G: ${ITprefix}.column3=name2
H:
I: ${ITprefix}.basedOn.name1=name
J: ${ITprefix}.basedOn.name2=name
```

This would result is the name being shown twice, as is seen to the right:



example1	Clock	$3000	example1
printer1	Printer	$3060	printer1
web1	NetscapeEx...	$3000	web1

The *label* property

The next step is to define the strings that will be used as the headers for the columns. The "label" resource controls this, and is defined as: *<Category name>.ItemTable.label.<column>*. (See the LABEL documentation for more details). For example, to add labels to the columns in the ItemTable, the resource file would get three new resources:

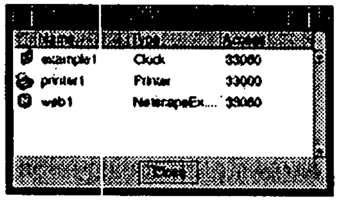
```
A: RHINO_EXAMPLE_CATEGORY=com.sgi.rhexamp.category.rhexampRhinoExampleCategory
B: ITprefix=${RHINO_EXAMPLE_CATEGORY}.ItemTable
C:
D: ${ITprefix}.column0=name
E: ${ITprefix}.column1=type
F: ${ITprefix}.column2=mode
G:
```

P22

COPY

```
H: $(ITprefix).label.name=Name
I: $(ITprefix).label.type=Type
J: $(ITprefix).label.mode=Access
```

Displaying the ItemTable now shows that the correct labels are displayed.



example1	Clock	33060
printer1	Printer	33000
web1	NetscapeEx...	33060

The *method* property

The next step is to choose what method the ItemTable should use to display the column. (*In this usage, "method" does not refer to a Java method, but rather to the typical English definition of the word.*) The "method" resource controls this, and is defined as: *<Category name>.ItemTable.method.<column>*. (See the METHOD documentation for more details) There are seven methods available:

1. toString

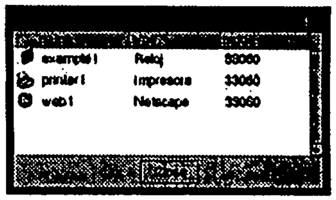
   The toString method is the default method, and is what the ItemTable implicitly uses to display the column if no method is specified in the properties file. The toString method calls Java's `toString` method on the value of the Attribute that is associated with the column (either by the "basedOn" property or the name of the column if no "basedOn" property is set). If this method is used, no additional resources are needed.
2. lookup

   The lookup method uses the value of the Attribute that is associated with the column (either by the "basedOn" property or the name of the column if no "basedOn" property is set) as a key to lookup a string in a table of values. This is useful for cases when the value of the Attributes comes from a limited set of possible values, and there is a mapping from the Attribute to some more easily understandable string. This is also useful when there will be a need to localize the text that gets displayed in the column. If using the "lookup" method, also provide "lookup" resources, which are defined as *<Category name>.ItemTable.lookup.<column>.<Attribute's value>* (See the LOOKUP documentation for more details), for each of the possible values of the Attribute.

   For example, to specify that the "type" column should use the lookup method, and should display the type in Spanish instead of English, include the following in the resource file:

```
A: $(ITprefix).method.type=lookup
B:
C: $(ITprefix).lookup.type.Printer=Impresora
D: $(ITprefix).lookup.type.Clock=Reloj
E: $(ITprefix).lookup.type.NetscapeExecutable=Netscape
```

In this case, the type of the Item will be displayed in it's Spanish equivalent:



example1	Reloj	33060
printer1	Impresora	33060
web1	Netscape	33060

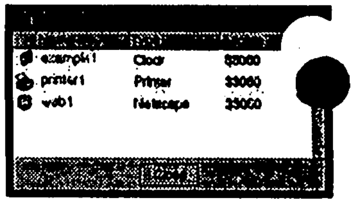
3. richText

   The `richText` method will display the string value of the Attribute just as the `toString` m does, but will display it as a link that launches an ItemView. This is generally used to show relationship between an Item in one Category and an Item in another Category, or to provide to launch ItemViews of the Items in the ItemTable.

   When using the `richText` method for a column, there must be two additional resources defined for each column. The first is the "category" resource, which is defined as *<Category name>.ItemTable.category.<column>* (See the CATEGORY documentation for more details). The second resource is the "selector" resource, which is defined as *<Category name>.ItemTable.selector.<column>* (See the SELECTOR documentation for more details). The "category" resource is a string that gives the package-qualifed name of the Category that the ItemView will use, and the "selector" resource names the Attribute whose value will be used as the selector of the Item the ItemView will show.

   The example only has one Category, so to demonstrate the `richText` method, consider making the "name" column contain links to launch the appropriate ItemView. The following lines would be added the resource file:

```
A: $(ITprefix).method.name=richText
B:
C: $(ITprefix).category.name=$(RHINO_EXAMPLE_CATEGORY)
D: $(ITprefix).selector.name=name
```



example1	Clock	33060
printer1	Printer	33060
web1	Netscape	33060

4. icon

   The `icon` method can be used to show the value of an Attribute as an icon. The value of the Attribute that is associated with the column (either by the "basedOn" property or the name of the column if no "basedOn" property is set) is turned into a string with the toString method, and that value is used to lookup the "icon" resource, which is defined as *<Category name>.ItemTable.icon.<column>.<Attribute's value>* (see the ICON documentation

P23

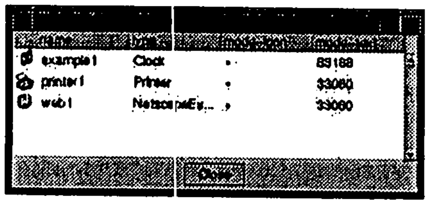
COPY

for more details). The "icon" resource should be the pathname of an icon to show in the table. If the "icon" resource is not found, then a default icon is used, which is defined as *<Category name>.ItemTable.icon.<column>* (See the DEFAULT_ICON documentation for more details). If neither the specific icon or the default icon icon is found, then no icon will be shown.

For example, to show an Icon that represents the "mode" Attribute, the following should be used as the resource file:

```
A: RHINO_EXAMPLE_CATEGORY=com.sgi.rhexamp.category.rhexampRhinoExampleCategory
B: ITprefix=$(RHINO_EXAMPLE_CATEGORY).ItemTable
C:
D: $(ITprefix).column0=name
E: $(ITprefix).column1=type
F: $(ITprefix).column2=mode-icon
G: $(ITprefix).column3=mode-text
H:
I: $(ITprefix).basedOn.mode-icon=mode
J: $(ITprefix).basedOn.mode-text=mode
K:
L: $(ITprefix).method.mode-icon=icon
M: $(ITprefix).icon.mode-icon.33188=/com/sgi/rhexamp/category/images/blue-ball.
N: $(ITprefix).icon.mode-icon.33060=/com/sgi/rhexamp/category/images/red-ball.g
O: $(ITprefix).icon.mode-icon=/com/sgi/rhexamp/category/images/yellow-ball.gif
```

The example to the right shows the ItemTable that will result. Note that because the mode needed to be displayed in two columns, once as an icon and once as text, it was necessary to add a forth column to the ItemTable and provide new names for the third and forth columns. Also notice the "basedOn" properties that tell ItemTable that the *mode* Attribute of the Item controls both columns.



```
example1   Clock          •   33188
printer1   Printer        •   33060
web1       NetscapeEx...  •   33060
```

5. stringRenderer

6. richTextRenderer

7. componentRenderer

It is sometimes the case that none of the three ways presented so far are adequate to display the state of the Item. Such cases can result when:

- O there is a need to synthesize two or more Attributes into a single value for display
- O Java code is needed to decode the Attribute (or Attributes) into a user-readable value
- O a special component is needed to display text
- O It is desired to show a label with color
- O the user would want to launch something other than an ItemView
- O anything else not permitted with the three predefined methods

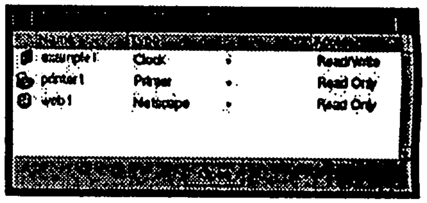
In any of these cases, use one of the renderer methods. These methods provide a chance to write a small piece of Java code that will control the display of the column. There are three types of renderers:

- O stringRenderer
- O richTextRenderer
- O componentRenderer

These renderers all use the same instance of ItemTableColumnRenderer to render the column. The renderers differ only in the type of Object that the renderer returns. In the case of the RhinoExample Category, the "mode-text" column uses a string renderer to convert the "mode" attribute of the Item into a user-readable string. See the RhinoExampleCategoryRenderers file for the example. Also see the Writing an ItemTableColumnRenderer section below about how to write an ItemTableColumnRenderer. Adding the following lines to the resource file tells the ItemTable to use the stringRenderer method for the "mode-text" column, and to use the `com.sgi.rhexamp.category.rhexampRhinoExampleCategoryRenderers` class as the ItemTableColumnRenderer. Lines "D" and "E" are resources that the ItemTableColumnl uses.

```
A: $(ITprefix).method.mode-text=stringRenderer
B: $(ITprefix).columnRenderer=$(RHINO_EXAMPLE_CATEGORY)Renderers
C:
D: $(ITprefix).modeStr.readWrite=Read/Write
E: $(ITprefix).modeStr.readOnly=Read Only
```

After adding the resources to use the string renderer, the ItemTable looks like:



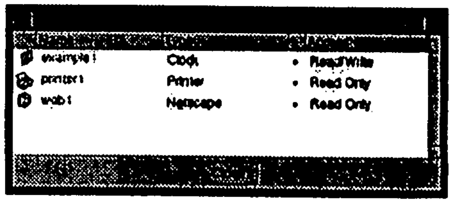
```
example1   Clock      •   Read/Write
printer1   Printer    •   Read Only
web1       Netscape   •   Read Only
```

The *width* property

To set the width of a column (in points), use the "width" property, which is defined as: *<Category name>.ItemTable.width.<column>*. (See the WIDTH documentation for more details). For example, to set the widths of the columns in the ItemTable, the resource file would get four new resources:

```
A: $(ITprefix).width.name=100
B: $(ITprefix).width.type=100
C: $(ITprefix).width.mode-icon=10
D: $(ITprefix).width.mode-text=100
```

After setting the widths, the ItemTable is as shown:



```
example1   Clock      •  Read/Write
printer1   Printer    •  Read Only
web1       Netscape   •  Read Only
```

P 24

COPY

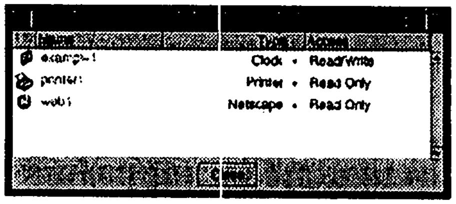
The *alignment* property

To control the alignment (justification) of the columns in the ItemTable, use the "alignment" property, which is defined as *<Category name>.ItemTable.alignment.<column>*. (See the ALIGNMENT documentation for more details). For example, to set the alignment of the columns in the ItemTable, the resource file would get two new resources:

```
A: $(ITprefix).alignment.type=right
B: $(ITprefix).alignment.mode-text=left
```

There are no alignment resources for the name or mode-icon columns because alignment is only available on columns using the toString, lookup, and stringRenderer methods.

After setting the alignment resources as shown above, the ItemTable looks like:



The *sort* property

To control the way the the ItemTable sorts a column, use the the "sort" property, which is defined as *<Category name>.ItemTable.sort.<column>*. (See the SORT documentation for more details). There are four sorting option available, and each of them work with particular methods:

- lexical (toString, lookup, richText, stringRenderer, or richTextRenderer methods)
- numeric (toString, lookup, richText, stringRenderer, or richTextRenderer methods)
- none (all methods)
- sorter (all methods)

The "lexical" sort is a alphanumeric sort that uses the `java.text.Collator.compare` method to compare Attributes. The "numeric" sort turns the Attributes into instances of java.lang.Integer and then performs a numeric sort. The "none" sort specifies that there is no sort order for a column. The "sorter" sort specifies that the ItemTable should call the `compareItemsForItemTable` method of the ItemTableColumnRenderer to compare Items. See the Writing an ItemTableColumnRenderer section below about how to write a ItemTableColumnRenderer. For example, to set the sort method of the columns in the ItemTable, the resource file would get four new resources:

```
A: $(ITprefix).sort.name=lexical
B: $(ITprefix).sort.type=lexical
C: $(ITprefix).sort.mode-icon=sorter
D: $(ITprefix).sort.mode-text=lexical
```

The *missing* property

Depending on the way that the server-side Category is written, there may be cases where a particular Attribute is missing from an Item. For example, consider that the Item can optionally contain the "type" Attribute. If the Item contains that Attribute, then the column should display the name using the `lookup` method as described above. Otherwise, the column should display some other string, such as "(Unknown)". For this situation, you can use the "missing" resource (defined as *<Category name>.ItemTable.missing.<column>*). The "missing" resource allows you to specify a string that will be displayed if an Attribute is missing from an Item. The "missing" resource can be used with the `toString`, `lookup`, or `richText` methods.

For example, to use the string "(Desconocido)" (Spanish for "Unknown") if the "type" Attribute is missing from the Item, add the following resource:

```
A: $(ITprefix).method.type=lookup
B:
C: $(ITprefix).lookup.type.Printer=Impresora
D: $(ITprefix).lookup.type.Clock=Reloj
E: $(ITprefix).lookup.type.NetscapeExecutable=Netscape
F: $(ITprefix).missing.type=(Desconocido)
```

Writing an ItemTableColumnRenderer

ItemTables use an instance of the ItemTableColumnRenderer interface to render columns that are using the `stringRenderer`, `richTextRenderer`, and `componentRenderer` methods. There is only one ItemTableColumnRenderer per ItemTable, so it must be able to handle all of the columns in the ItemTable that are using a renderer method. Write a class that implements the ItemTableColumnRenderer interface, and place it in the product's "category" package. (The file can actually be placed anywhere, but the "category" package is one logical place.) Tell the ItemTable how to find the class by naming it in the property file with the "columnRenderer" property, which is defined as *<Category name>.ItemTable.columnRenderer* (see the COLUMN_RENDERER documentation for more info). For example, the RhinoExampleCategory has a class `com.sgi.rhexamp.category.rhexampRhinoExampleCategoryRenderers` that implements the ItemTableColumnRenderer interface, and so the following line is included in the Category's resour file:

```
A: $(ITprefix).columnRenderer=$(RHINO_EXAMPLE_CATEGORY)Renderers
```

If a column uses a renderer method, but no ItemTableColumnRenderer is defined with the "columnRenderer" property, then the ItemTable will attempt to load a class with the name *<Category Name>ColumnRenderer*. For example, for the rhinoExampleCategory, it would attempt to load the class `com.sgi.rhexamp.category.rhexampRhinoExampleCategoryColumnRenderer`. If the "columnRenderer" resource is not specified and the *<Category Name>ColumnRenderer* class is not found, then ItemTable will throw an assertion.

The ItemTableColumnRenderer has four methods that must be implemented: See the documentation for ItemTableColumnRenderer about the specifics of each method.

- `public String getStringForCellOfItemTable(Item item, String columnName,`

P25

```
ItemTableContext context)
```

- `public String getRichTextForCellOfItemTable(Item item, String columnName, ItemTableContext context);`
- `public JComponent getComponentForCellOfItemTable(Item item, String columnName, ItemTableContext context)`
- `public int compareItemsForItemTable(Item item1, Item item2, String columnName);`

When the ItemTable requires that a cell be rendered, it will call one of the get*ForCellOfItemTable methods, depending on the type of renderer being used.

For the *stringRenderer* method, the ItemTable will call the `getStringForCellOfItemTable` method, and the method should compute the String to display and return it.

For the *richTextRenderer* method, the ItemTable will call the `getRichTextForCellOfItemTable` method, and the method should compute the String of HTML to display in a RichTextComponent and return it. To construct a URL that will launch an ItemView, use the `createURLToLaunch` method of ItemView.

For the *componentRendrer* method, the ItemTable will call the `getComponentForCellOfItemTable`, and the method should return a Component that the ItemTable should display in the appropriate cell.

The `compareItemsForItemTable` method is used to sort the ItemTable based on a column that is using the "sorter" method of sorting. The ItemTable will pass two Items and the name of the column to the method, and the method should return an integer representing which of the Items should come first in the sorted list. See the ItemTableColumnRenderer documentation for more information about these methods.

Controlling the Icon displayed for an Item in the ItemTable

The ItemTable does not directly control the Icon that is displayed. The Icon is generated by the ResourceBasedIconRenderer. See the tutorial on using ResourceBasedIconRenderer for details on how to control the Icon.

Controlling the title of the ItemTable

The ItemTable does not directly control the title that is used (as displayed on the window's title bar). The Title is generated by the ResourceBasedNameRenderer. See the tutorial on using ResourceBasedNameRenderer for details on how to control the title.

How to launch ItemTables

To view an ItemTable from the command line, type:

```
%> java com.sgi.sysadm.manager.RunItemTable <Category Name>
```

For example, to launch an ItemTable for Category "BarCategory", where the ItemTable's resource file is in `/com/sgi/myProduct/category` (relative to classpath), type:

```
%> java com.sgi.sysadm.manager.RunItemTable com.sgi.myProduct.category.BarCategory
```

To launch a no-code ItemTable, omit the name of the package:

```
%> java com.sgi.sysadm.manager.RunItemTable BarCategory
```

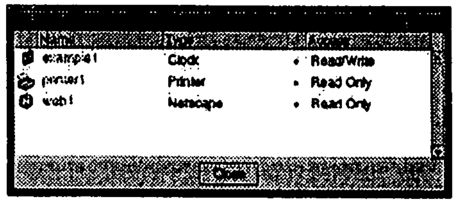
To programmatically launch an ItemTable, use one of two methods: To launch an ItemTable in a new frame (called an ItemTableFrame), use the `launchItemTableFrame` method in ItemTableFrame. The `launchItemTableFrame` method takes a ItemTableLaunchRequestEvent, which encapsulates all the information about which ItemTable to launch. For example:

```
ItemTableFrame.launchItemTableFrame(
    new ItemTableLaunchRequestEvent(this,
                                    "com.sgi.myProduct.category.BarCategory"),
    new UIContext());
```

To embed an ItemTable in another component, create an ItemTable with the `createItemTable` method of ItemTable, set Category to display with the `setCategory` method, then call `getItemTablePanel` on ItemTable to get a panel that contains the ItemTable. For example:

```
ItemTable it = ItemTable.createItemTable(_hostContext,
                                 "com.sgi.myProduct.category.BarCategor
it.setCategory(_hostContext.getCategory("BarCategory"));
_panel.add(it.getItemTablePanel());
```

Typical Resource File for an ItemTable



Name	Type	Access
example1	Clock	• Read/Write
printer1	Printer	• Read Only
web1	Netscape	• Read Only

```
# Set up some macros to use in this resource file.  See the
# ResourceStack documentation for more about macros.
RHINO_EXAMPLE_CATEGORY=com.sgi.rhexamp.category.rhexampRhinoExampleCategory
```

P26

```
ITprefix=$(RHINO_EXAMPLE_CATEGORY).ItemTable

# Define the columns to displayed.  Call them 'mode', 'type',
# 'mode-icon', and 'mode-text'.
$(ITprefix).column0=name
$(ITprefix).column1=type
$(ITprefix).column2=mode-icon
$(ITprefix).column3=mode-text

# Tells the ItemTable which Attributes of the Item to use to show the
# columns.  It is not necessary to set a resource for mode-text because
# it's using a 'renderer' method, and basedOn is not used for the
# 'renderer' methods.
$(ITprefix).basedOn.name=name
$(ITprefix).basedOn.type=type
$(ITprefix).basedOn.mode-icon=mode

# Sets the method that the ItemTable will use to display the four columns
$(ITprefix).method.name=richText
$(ITprefix).method.type=lookup
$(ITprefix).method.mode-icon=icon
$(ITprefix).method.mode-text=stringRenderer

# Additional resources that are necessary because the 'mode-icon'
# column is using the 'icon' method.
$(ITprefix).icon.mode-icon.33188=/com/sgi/rhexamp/category/images/blue-ball.gif
$(ITprefix).icon.mode-icon.33060=/com/sgi/rhexamp/category/images/red-ball.gif
$(ITprefix).icon.mode-icon=/com/sgi/rhexamp/category/images/yellow-ball.gif

# Additional resources that are necessary because the 'type' column
# is using the 'lookup' method.
$(ITprefix).lookup.type.Printer=Printer
$(ITprefix).lookup.type.Clock=Clock
$(ITprefix).lookup.type.NetscapeExecutable=Netscape

# Additional resources that are necessary because the 'name' column
# is using a 'richText' method.
$(ITprefix).category.name=$(RHINO_EXAMPLE_CATEGORY)
$(ITprefix).selector.name=name

# Sets the labels that will be used for the columns.
$(ITprefix).label.name=Name
$(ITprefix).label.type=Type
$(ITprefix).label.mode-icon=
$(ITprefix).label.mode-text=Access

# Sets the widths of the columns
$(ITprefix).width.name=100
$(ITprefix).width.type=100
$(ITprefix).width.mode-icon=10
$(ITprefix).width.mode-text=100

# Sets the alignment that will be used for the columns.  There are
# no alignment resources for the name or mode-icon columns because
# alignment is only available on columns using the toString, lookup, and
# stringRenderer methods.
$(ITprefix).alignment.type=left
$(ITprefix).alignment.mode-text=left

# Sets the type of sort that will be used for the columns.
$(ITprefix).sort.name=lexical
$(ITprefix).sort.type=lexical
$(ITprefix).sort.mode-icon=sorter
$(ITprefix).sort.mode-text=lexical

# Tells the ItemTable what class to use as the ItemTableColumnRenderer.
$(ITprefix).columnRenderer=$(RHINO_EXAMPLE_CATEGORY)Renderers

# Resources used by the ItemTableColumnRenderer.
$(ITprefix).modeStr.readWrite=Read/Write
$(ITprefix).modeStr.readOnly=Read Only
```

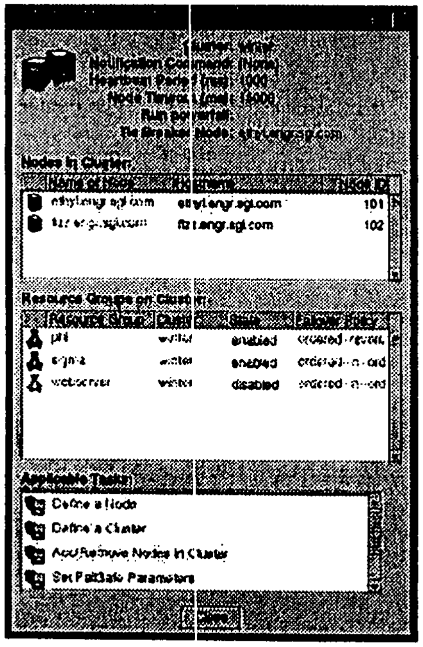
P27

COPY

Putty-8053

How to write a Rhino ItemView

Table of Contents

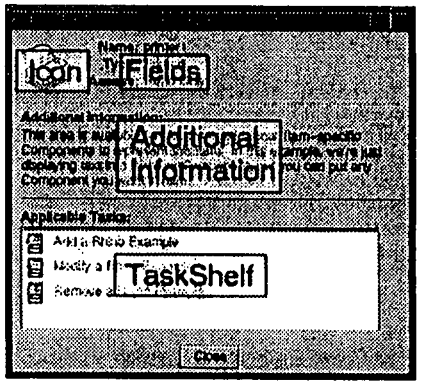


Introduction

This document is a reference for SGI software engineers who will be writing ItemViews for Rhino applications. An ItemView in the Rhino Architecture is a UI Component that displays all relevant information about a particular Item. The ItemView is the user's main source of information about the attributes of an Item, which include both static and dynamic information.

Overview of the ItemView's Sections

Shown below is a picture of an ItemView, with the different sections labeled.



Icon	Shown in the upper left corner of the ItemView. Typically, the Icon represents the type of Item being viewed, and additionally the state of the Item
Fields	Shown in the upper right corner of the ItemView. T section is divided into tw columns, the left for the nam of the field, and the right for the value of the field. The fields section is designed to show information about the Item that can be represented by fairly short Strings
Additional Info section	This section occupies the center of the ItemView. It is an optional section. This section is designed to show information about the Item that can't easily be represented as a single line of text. Examples include ItemTables, graphs, or additional icons. Any Java component can be shown here. If there are no components to show in this section, then the ItemView will not show the Additio Info section.
TaskShelf section	This section occupies the bottom of the ItemView. It shows a TaskShelf containing Tasks that can operate on the displayed Item in the Item's current state

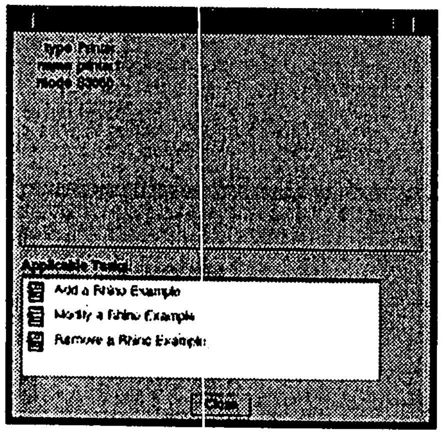
Before you begin

P28

Before beginning to create an ItemView for a particular Category, it is necessary to understand the names and terms that the Rhino infrastructure uses in relation to Categories. See the The Names of Categories on the Client and on the Server documentation for more information.

How to create an ItemView for a particular Category

No-Code ItemViews



While in the early stages of writing Categories, it may be desirable to show an ItemView that shows all of the Attributes of an Item. The ItemView supports this idea by means of a "no-code" ItemView. This version of an ItemView is not designed for use in a shipping Rhino application, but can be of great assistance while investigating the Rhino Infrastructure or for giving preliminary demos. No code or resource files need to be written to use the "no-code" ItemView - it can be launched as soon the server side Categories have been written and the Rhino infrastructure has been installed on the client. To turn the ItemView into a shippable ItemView, it is necessary to provide resources that describe the way that the Attributes of the Item are to be displayed. The rest of this document will describe how to accomplish this. To launch a "no-code" ItemView, follow the instructions in the section titled How to launch ItemViews. An example of the "no-code" ItemView for the RhinoExampleCategory is shown to the right.

Analyze Item's attributes

Before writing any code or resource files for the ItemView, begin by analyzing the information that needs to be displayed. Divide the information into two groups: information that will go in the Fields section, and information that will go in the Additional Information section. While there are no absolute rules about what kind of information goes where, here are some suggestions on how to divide the information:

- The Fields section is best suited for displaying short text strings. The Additional Information section has the ability to display larger components.
- It is suggested that information that defines the identity of the Item be in the Fields section where it will be easy to find. Secondary information, such as information about relationships between the Item and other Items, can be in the Additional Information section and interested users can take the time to locate it.
- It may be desireable to place information that is static or changes infrequently in the Fields section, and put dynamic information that changes often during the normal operation of the system in the Additional Information section.
- Don't put too many pieces of information in the Fields section. More than about 7 lines will make the ItemView hard to read. Try to split the information into smaller sets that are logically and semantically grouped. Put the most important sets of information in the Fields section and put the rest in the Additional Information section.
- In each section, order the information by importance, with the most important information at the top.
- In some cases, it makes sense to break these guidelines to group common pieces of informa[illegible] to give the ItemView an appealing layout that's easy to understand.

All of the information in a RhinoExampleCategory Item probably belongs in the Fields section, but imagine that the Printer type of Item also had a list of print jobs. Since the list could be quite long, using a comma separated list would not be a practical solution. In this case, it might work to use a JList to implement a scrollable list to display all of the print jobs. This component would be displayed in the Additional Information section).

Customizing the Fields of the ItemView

There are several types of properties that control the look of the ItemView. The properties (in the order that they are described) are:

1. The *field* properties - Define names of the fields.
2. The *basedOn* properties - Tell the ItemView which Attributes correspond to particular fields.
3. The *label* properties - Provide the labels the ItemView will use for the fields.
4. The *method* properties - Specify the manner in which the ItemView will use the Item's Attributes to fill in the field.

The *field* Properties

The pieces of the Item's information that are displayed in the Fields section are completely contr[illegible] a resource file. The most fundamental resources are those that give names to the fields that will be displayed. These names identify the fields so that other resources can refer to particular fields. The resources follow the form *<Category name>.ItemView.field<n>*, where <Category name> is the name of the Category (see the FIELDS documentation for more info), and <n> represents integers starting at 0 that represent order in which the fields should be displayed. For example, the resource file that controls the RhinoExampleCategory contains the following lines (the letters in the first column are for reference purposes only):

```
A: com.sgi.rhexamp.category.rhexampRhinoExampleCategory.ItemView.field0=name
B: com.sgi.rhexamp.category.rhexampRhinoExampleCategory.ItemView.field1=type
C: com.sgi.rhexamp.category.rhexampRhinoExampleCategory.ItemView.field2=mode
```

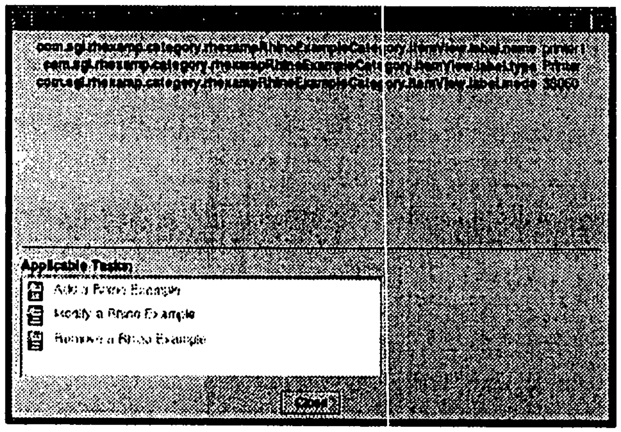
Because the first part of each line is identical, it is common to use macros to shorten the lines of the

P29

COPY

resource file and to make the file easier to read. An example of the same resources using macros is shown below.

```
A: RHINO_EXAMPLE_CATEGORY=com.sgi.rhexamp.category.rhexampRhinoExampleCategory
B: IVprefix=$(RHINO_EXAMPLE_CATEGORY).ItemView
C:
D: $(IVprefix).field0=name
E: $(IVprefix).field1=type
F: $(IVprefix).field2=mode
```



The three "field" resources (D - F) define the names of the fields and the order in which the fields will be displayed in the ItemView. In this example, the names of the fields correspond exactly with the names of the Attributes in the Item that will be displayed in the field. By naming the fields in this manner, the ItemView can use default behavior and automatically associate the correct Attribute with the field. It is also possible to give the fields names that are not the same as the names of Attributes. In that case, it may be necessary to use the "basedOn" property (defined below) to tell the ItemView which Attribute is associated with a field.

Running an ItemView with the 5 lines described above in the resource file will result in an ItemView that is shown on the right. Notice that the order of the fields is "name", "type", and then "node", which is as specified in the resource file. ItemView has used a default label for each of the fields. Information about how to customize the label is described below. The ItemView is using the default "toString" method (methods are described below). This is the simplest method, and uses the results of calling Java's `toString` method on the value of the Attribute.

The *basedOn* Properties

In the example resource file shown above, the names of the fields were defined to be the same as the Item's Attributes that they represented. This allowed the ItemView to automatically show the value of the Attribute in the field. It is sometimes desirable to use different names for the fields than the Attributes that they represent. This can make the resource file more readable or can be required because there may not be a one to one correspondence between the Attributes in the Item and the fields that are displayed.

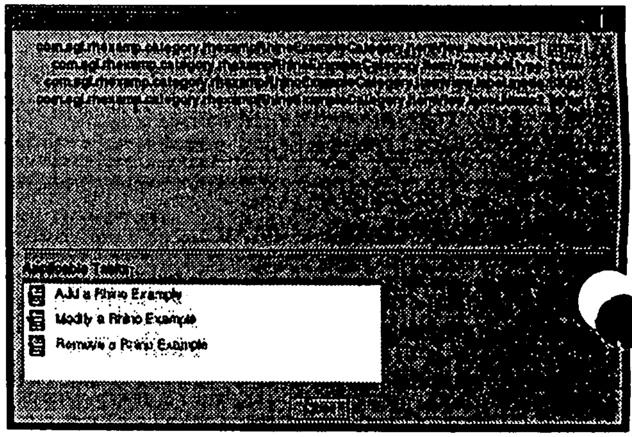
If a field is given a name that does not correspond to the name of an Attribute, the "basedOn" property is used to tell the ItemView which Attribute the field represents. The "basedOn" resources are defined as: *<Category name>.ItemView.basedOn.<field>*, where <Category name> is the name of the Category, and <field> is the name of a field. (See the BASED_ON documentation for more info).

The renderer method (as described below) does not require that the field be associated with a particular Attribute. When using this method, it is not necessary to specify the "basedOn" property even if the name of the field does not correspond to an Attribute. All the other methods, including the default "toString" method, require that the field be associated with a particular Attribute of the Item.

For example, suppose that the "name" Attribute should be displayed twice, once at the beginning of the list, and once at the end. A resource file as follows would do just that:

```
A: RHINO_EXAMPLE_CATEGORY=com.sgi.rhexamp.category.rhexampRhinoExampleCategory
B: IVprefix=$(RHINO_EXAMPLE_CATEGORY).ItemView
C:
D: $(IVprefix).field0=Name1
E: $(IVprefix).field1=type
F: $(IVprefix).field2=node
G: $(IVprefix).field3=Name2
H:
I: $(IVprefix).basedOn.Name1=name
J: $(IVprefix).basedOn.Name2=name
```

This would result is the name being shown twice, as is seen to the right:



The *label* Properties

The next step is to define the strings that will be used as the labels for the fields. The "label" resources are defined as: *<Category name>.ItemView.label.<field>.label*. (See the LABEL documentation for more info). Optionally, another resource can be specified that gives the name of a glossary entry that will be displayed if the user clicks on the label. This resource is defined as: *<Category name>.ItemView.label.<field>.glossary*, and if this resource is defined, then the label will

P30

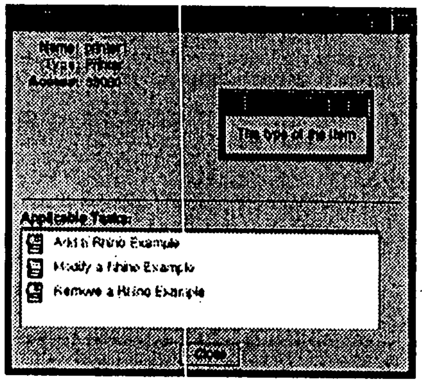
COPY

appear blue.

For example, define labels for the example ItemView, the following properties would be added to the resource file:

```
A: RHINO_EXAMPLE_CATEGORY=com.sgi.rhexamp.category.rhexampRhinoExampleCategory
B: IVprefix=$(RHINO_EXAMPLE_CATEGORY).ItemView
C:
D: $(IVprefix).field0=name
E: $(IVprefix).field1=type
F: $(IVprefix).field2=mode
G:
H: $(IVprefix).label.name.label=Name:
I: $(IVprefix).label.type.label=Type:
J: $(IVprefix).label.type.glossary=glossary.Type
K: $(IVprefix).label.mode.label=Access:
L:
M: glossary.Type = The type of the Item
```

Displaying the ItemView now shows that the desired labels are displayed. Notice that the "type" label is displayed as a link, and the picture shows the glossary window that results when the user clicks on the link.



The *method* Properties

The next step is to choose what method the ItemView should use to display the field. (*In this usage, "method" does not refer to a Java method, but rather to the typical English definition of the word*) The "method" resource controls this, and is defined as: *<Category name>.ItemView.method.<field>* (see the METHOD documentation for more info). Four methods are available:

1. toString

The toString method is the default method, and is what the ItemView implicitly uses to display the field if no method is specified in the properties file. The toString method calls Java's `toString` method on the value of the Attribute that is associated with the field (either by the "basedOn" property or the name of the field if no "basedOn" property is set). If this method is used, no additional resources are needed. For example, to make explicit the fact that the "name" field should use the toString method, include the following in the resource file:

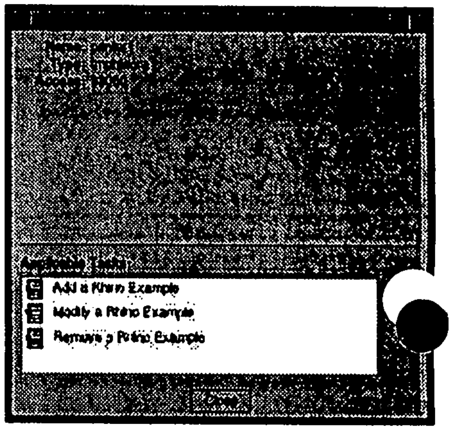
```
$(IVprefix).method.name=toString
```

2. lookup

The lookup method uses the value of the Attribute associated with the field (either by the "basedOn" property or the name of the field if no "basedOn" property is set) as a key to lookup a string in a table of values. This is good for cases when the value of the Attributes comes from a limited set of possible values, and there is a mapping from the Attribute's value to some more easily understandable string. This method is also good when there is a need to localize the text that gets displayed in the field. If the "lookup" type is used, additional "lookup" resources (defi *<Category name>.ItemView.lookup.<field>.<Attribute's value>*) should also be provided each of the possible values of the Attribute. For example, to specify that the "type" field should use the lookup method, and should display the type in Spanish instead of English, include the following in the resource file:

```
A: $(IVprefix).method.type=lookup
B:
C: $(IVprefix).lookup.type.Printer=Impresora
D: $(IVprefix).lookup.type.Clock=Reloj
E: $(IVprefix).lookup.type.NetscapeExecutable=Netscape
```

In this case, the type of the Item will be displayed in it's Spanish equivalent:



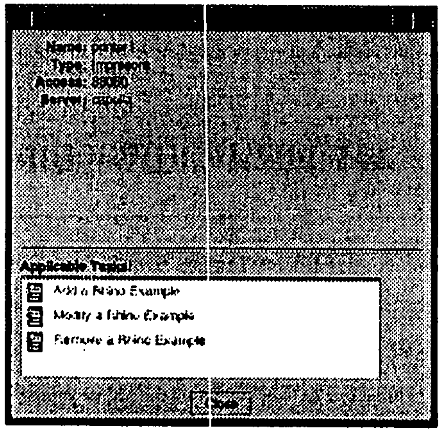
3. richText

The `richText` method will display the string value of the Attribute just as the `toString` method does, but will display it as a link that launches an ItemView. This is generally used to show the relationship between an Item in one Category and an Item in another Category. The example used in this document has one Category, but consider the case where each of the Items in the RhinoExample category had an Attribute in it that specifies the server on which the Item was running. Assume also that there is a second Category, "rhexampServerCategory" with server

P31

Items. Consider that the RhinoExample Item has an Attribute with the name "server" that is the name of the server that the RhinoExample is running on, and another Attribute "server_selector" which is the selector of the server in the "rhexampServerCategory" category. (In many cases, the name of the server would be the same as the selector of the server. In that case, substitute "server" for "server_selector" in the following example.) To show a link to the appropriate server from the RhinoExample ItemView, the following would be added to the Resource File:

```
A: $(IVprefix).field3=server
B: $(IVprefix).label.server=Server:
C:
D: $(IVprefix).method.server=richText
E: $(IVprefix).selector.server=server_selector
F: $(IVprefix).category.server=rhexampServerCategory
```



4 renderer

It is sometimes the case that none of the three ways presented so far are adequate to display the state of the Item. Such cases can result when:

- O there is a need to synthesize two or more Attributes into a single value for display
- O Java code is needed to decode the Attribute (or Attributes) into a user-readable value
- O A special component is needed to display text
- O It is desired to show a label with color
- O The user would want launch something other than an ItemView
- O any thing else not permitted with the three predefined methods

In any of these cases, the renderer method should be used. This method provides a chance to write a small piece of Java code that will control the display of the field. In the case of the RhinoExample Category, the renderer converts the numeric "mode" Attribute into text that is displayed to the user. For example, the mode "33060" is displayed as 'Read Only". See the RhinoExampleCategoryRenderers file for this example.

The *missing* properties

Depending on the way that the server-side Category is written, there may be cases where a particular Attribute is missing from an Item. For example, consider that the Item can optionally contain the "type" Attribute. If the Item contains that Attribute, then the ItemView should display the name using the `lookup` method as described above. Otherwise, the ItemView should display some other string, such as "(Unknown)". For this situation, you can use the "missing" resource (defined as *<Category name>.ItemView.missing.<field>*). The "missing" resource allows you to specify a string that will be displayed if an Attribute is missing from an Item. The "missing" resource can be used with the `toString`, `lookup`, or `richText` methods.

For example, to use the string "(Desconocido)" (Spanish for "Unknown") if the "type" Attribute is missing from the Item, add the following resource:

```
A: $(IVprefix).method.type=lookup
B:
C: $(IVprefix).lookup.type.Printer=Impresora
D: $(IVprefix).lookup.type.Clock=Reloj
E: $(IVprefix).lookup.type.NetscapeExecutable=Netscape
F: $(IVprefix).missing.type=(Desconocido)
```

Writing an ItemViewFieldRenderer

ItemViews use an instance of the ItemViewFieldRenderer interface to render fields that use the `renderer` method. There is only one ItemViewFieldRenderer per ItemView, so it must be able to handle all of the fields in the ItemView that are using the renderer method. A class should be written that implements the ItemViewFieldRenderer interface, and placed in the product's "category" package. (The file can actually be placed anywhere, but the "category" package is one logical place). Tell the ItemView how to find the class by naming it in the property file with the "fieldRenderer" property, which is defined as *<Category name>.ItemView.fieldRenderer* (see the FIELD_RENDERER documentation for more info). For example, the RhinoExampleCategory (whose full name is `com.sgi.rhexamp.category.rhexampRhinoExampleCategory`) has a class `com.sgi.rhexamp.category.rhexampRhinoExampleCategoryRenderers` that implements the ItemViewFieldRenderer interface, and so the following line is included in the Category's resourc

```
A: $(IVprefix).fieldRenderer=$(RHINO_EXAMPLE_CATEGORY)Renderers
```

If a field uses the `renderer` method, but no ItemViewFieldRenderer is defined with the "fieldRenderer" property, then the ItemView will attempt to load a class with the name *<Category Name>FieldRenderer*. For example, for the RhinoExampleCategory, it would attempt to load the class `com.sgi.rhexamp.category.rhexampRhinoExampleCategoryFieldRenderer`. If the "fieldRenderer" resource is not specified and the *<Category Name>FieldRenderer* class is not found, then ItemView will throw an assertion.

The ItemViewFieldRenderer has five methods that must be implemented. See the documentation for ItemViewFieldRenderer about the specifics of each method.

- `public void initializeFieldRenderer(ItemViewContext ivc, ItemViewController`

P32

```
controller);
```

- `public Component getComponentForField(String field);`
- `public void renderFields(Item item);`
- `public void renderFieldsAgain(Item item);`
- `public void renderFieldsBlank();`

The sequence that the methods will be called in is as follows:

1. initializeFieldRenderer
2. getComponentForField (once for each field using the renderer)
3. renderFields
4. renderFieldsAgain (zero or more times)
5. renderFieldsBlank
6. repeat from step 3 (only if ItemView is used to display another Item)

The `initializeFieldRenderer` method is responsible for initializing the renderer. The ItemView calls the `getComponentForField` method once for each field that is using the ItemViewFieldRenderer. The ItemView passes in the name of the field. and the renderer passes back the component that the ItemView should use to display the field. When the ItemView obtains an Item, it passes the Item to `renderFields`. At this time, the renderer should use the Item to fill in the information that the components are displaying. If the Item changes, the ItemView calls `renderFieldsAgain`, passing the Item. The renderer should update the components to show the current state. If the Item is deleted, or the ItemView is disposed, the ItemView will call `renderFieldsBlank`. The renderer should remove any state information from the component, and prepare itself to be garbage collected. In the case that the Item reappears, or the ItemView is set to display another Item, the sequence repeats from step 3 (`renderFields`).

Writing an ItemViewAdditionalInfoRenderer

The additional info area of the ItemView is completely at the discretion of the programmer, and Java code must be written to display anything. ItemViews use an instance of ItemViewAdditionalInfoRenderer to render the additional info section. Write a class that implements the ItemViewAdditionalInfoRenderer interface, and place it in the product's "category" package. (The file can actually be placed anywhere, but the "category" package is one logical place). Tell the ItemView how to find the class by naming it in the property file with the "additionalInfoRenderer" property, which is defined as *<Category name>.ItemView.additionalInfoRenderer* (see the ADDITIONAL_INFO_RENDERER documentation for more info). For example, the RhinoExampleCategory (whose full name is `com.sgi.rhexamp.category.rhexampRhinoExampleCategory`) has a class `com.sgi.rhexamp.category.rhexampRhinoExampleCategoryRenderers` that implements the ItemViewAdditionalInfoRenderer interface, and so the following line is included in the Category's resource file:

```
A: ${IVprefix}.additionalInfoRenderer=${RHINO_EXAMPLE_CATEGORY}Renderers
```

If no ItemViewAdditionalInfoRenderer is defined with the "additionalInfoRenderer" property, then the ItemView will attempt to load a class with the name *<Category Name>AdditionalInfoRenderer*. For example, for the rhinoExampleCategory, it would attempt to load the class `com.sgi.rhexamp.category.rhexampRhinoExampleCategoryAdditionalInfoRenderer`. If the "addtionalInfoRenderer" resource is not specified and the *<Category Name>AdditionalInfoRenderer* class is not found, then ItemView will not display anything in the "Additional Information" section.

The API for the ItemViewAdditionalInfoRenderer is almost identical to that of the ItemViewFieldRenderer. In this case, there are four methods that must be implemented:

- `public void initializeAdditionalInfoRenderer(LabelComponentPanel panel, ItemViewContext ivc, ItemViewController controller);`
- `public void renderInfo(Item item);`
- `public void renderInfoAgain(Item item);`
- `public void renderInfoBlank();`

The sequence that the methods will be called in is as follows:

1. initializeAdditionalInfoRenderer
2. renderInfo
3. renderInfoAgain (zero or more times)
4. renderInfoBlank
5. repeat from step 2 (only if ItemView is used to display another Item)

The `initializeAdditionalInfoRenderer` method is responsible for initializing the renderer and preparing it for use. The ItemView passes the method a LabelComponentPanel that the renderer should add its components to. When the ItemView receives an Item, it passes the `renderInfo` method the Item, and the renderer should update the components on the panel as it wishes. If the Item changes its state then the ItemView will call `renderInfoAgain` and the renderer should update all of the components to show the current state. If the Item is deleted or the ItemView is disposed, then the ItemView will call renderInfoBlank, and in response the renderer should update the components to not show any state and prepare to be garbage collected. In the case that the Item reappears, or the ItemView is set to display another Item, the sequence repeats from step 2 (`renderInfo`).

Controlling the Icon displayed for an ItemVie

The ItemView does not directly control the Icon that is displayed. The Icon is generated by the ResourceBasedIconRenderer. See the tutorial on using ResourceBasedIconRenderer for details on how to control the Icon.

Controlling the title of the ItemView (as displayed on the window's title bar)

The ItemView does not directly control the title that is used. The Title is generated by the ResourceBasedNameRenderer. See the tutorial on using ResourceBasedNameRenderer for details on

P33

COPY

how to control the title.

Controlling the TaskShelf on the ItemView

The ItemView does not directly control the Tasks shown in the TaskShelf. The Tasks are generated by the TaskRegistry.

How to launch ItemViews

To view an ItemView from the command line, type:

```
%>
java com.sgi.sysadm.manager.RunItemView <package qualified Category name> <Item sele
```

For example, to launch an ItemView of the Item "foo" in Category "BarCategory", where the ItemView's resource file is in `com/sgi/myProduct/category` (relative to classpath), type:

```
%> java com.sgi.sysadm.manager.RunItemView com.sgi.myProduct.category.BarCategory
foo
```

To launch a "no-code" ItemView, pass the Category selector instead of the fully qualified name:

```
%> java com.sgi.sysadm.manager.RunItemView BarCategory foo
```

A "no-code" ItemView will also be displayed if no resources corresponding to the Category are found.

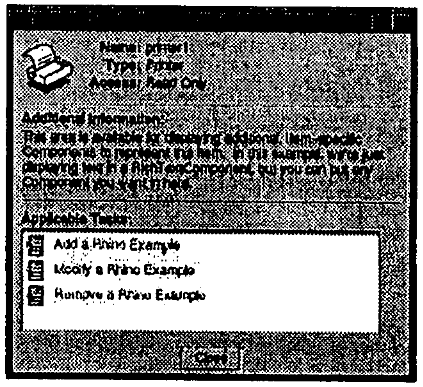
To programmatically launch an ItemView, use one of two methods: To launch an ItemView in a new frame (called an ItemViewFrame), use the `launchItemViewFrame` method in ItemViewFrame. The `launchItemViewFrame` method takes a ItemViewLaunchRequestEvent, which encapsulates all the information about which ItemView to launch. For example:

```
ItemViewFrame.launchItemViewFrame(
    new ItemViewLaunchRequestEvent(this,
                                   "com.sgi.myProduct.category.BarCategory",
                                   "foo"),
    new UIContext());
```

To embed an ItemView in another component, create an ItemView with the createItemView method of ItemView, set Item to display with the `setItem` method, then call `getPanel` on ItemView to get a panel that contains the ItemView. For example:

```
ItemView iv = ItemView.createItemView(_hostContext,
                                      "com.sgi.myProduct.category.BarCategory")
iv.setItem("foo");
_panel.add(iv.getPanel());
```

Typical Resource File for an ItemView



```
# Set up some macros to use in this resource file.  See the
# ResourceStack documentation for more about macros.
RHINO_EXAMPLE_CATEGORY=com.sgi.rhexamp.category.rhexampRhinoExampleCategory
IVprefix=${RHINO_EXAMPLE_CATEGORY}.ItemView
ITprefix=${RHINO_EXAMPLE_CATEGORY}.ItemTable


# Set the width and height of the ItemView.  See the PANEL_WIDTH and
#  PANEL_HEIGHT documentation for more information.
ItemViewPanel.width=333
ItemViewPanel.height=260

# Set up the three fields.  Call the fields "name", "type", and "mode".
${IVprefix}.field0=name
${IVprefix}.field1=type
${IVprefix}.field2=mode

# Tell the ItemView which Attributes of the Item to use to show the
# appropriate field.  It is not necessary to set the "basedOn" resource
# for a field that is using the "renderer" method, which is why there is
# no "${IVprefix}.basedOn.mode" resource.  In this case, the next two
# lines are unnecessary, because the name of the Attribute is the same
# as the field.  They are included here to make the resource file easier
# to understand and for illustration purposes.
${IVprefix}.basedOn.name=name
${IVprefix}.basedOn.type=type

# Sets the labels to be used for the three fields.
${IVprefix}.label.name.label=Name:
${IVprefix}.label.type.label=Type:
${IVprefix}.label.mode.label=Access:

```

p 34

COPY

```
# Sets the method that the ItemView will use to display the
# three fields.
$(IVprefix).method.name=toString
$(IVprefix).method.type=lookup
$(IVprefix).method.mode=renderer

# Resources necessary because the 'type' field is using the
# 'lookup' method.  See the description of the lookup method for more informatio
$(IVprefix).lookup.type.Printer=Printer
$(IVprefix).lookup.type.Clock=Clock
$(IVprefix).lookup.type.NetscapeExecutable=Netscape

# Tells the ItemView what classes to use as the
# ItemViewFieldRenderer and the ItemViewAdditionalInfoRenderer.  In this
# case, both renderers are in the same class, but this is not
# necessarily the case.
$(IVprefix).fieldRenderer=$(RHINO_EXAMPLE_CATEGORY)Renderers
$(IVprefix).additionalInfoRenderer=$(RHINO_EXAMPLE_CATEGORY)Renderers

# Resources specific to the AdditionalInfoRenderer.  The
# AdditionalInfoRenderer has access the the ResourceStack, so this file
# is a good place to put resources that control the
# ItemViewAdditionalInfoRenderer or ItemViewFieldRenderer.  The names of
# the resources are specific to the code that is written in the
# renderers.
ItemView.AdditionalInfo.marginLeft=0
ItemView.AdditionalInfo.marginTop=0
ItemView.AdditionalInfo.marginBottom=0
ItemView.AdditionalInfo.marginRight=0
ItemView.AdditionalInfo.layoutType=vertical
ItemView.AdditionalInfo.label=Additional Information:
ItemView.AdditionalInfo.text=This area is available for displaying \
 additional, item-specific Components to represent this item.  In this \
 example, we're just displaying text in a RichTextComponent, but you can \
 put any Component you want in here.
```

P35

COPY

Putty - 8034

mkrhinoism - The Rhino ISM Generator

Rhino provides a self-building example ISM (independent software module) to help you get started using Rhino to create your application. Follow these instructions to use the example ISM:

1. Install the example ISM (sysadm_noship.sw.ismtools) along with required prereqs (default sysadm_base.sw):

```
# inst -f ethyl.engr.sgi.com:/dist/rhino1.0
```

2. Run the mkrhinoism command, which will launch a Rhino-based Task. This Task guides you through the steps to create an actual Rhino application ISM complete with code containing stubs you can augment to become your actual Rhino application.

```
% rehash; mkrhinoism
```

Here are some example inputs:

Category:	Volume
ISM Abbreviation:	xlvgui
Product Name:	xlv
Directory:	/usr/people/rcu/xlvgui

When you've specified these bits of information, press the **OK** button. A status dialog will appear so you can watch as files are copied from bonnie and the build script is created.

The selections files that `mkrhinoism` uses for creating a ROOT and TOOLROOT are:
http://ethyl.engr.sgi.com/rhino/ismtools/root.selections
http://ethyl.engr.sgi.com/rhino/ismtools/troot.selections

For information on the directory structure of the ISM generated by mkrhino ism, see Rhino ISM Directory Structure.

3. Follow the instructions in the ResultView that appears to build your new ISM. Using the above example, the instructions would be

```
% cd /usr/people/rcu/xlvgui
% ./makeme
```

4. Install the images built in step 3 by running the following commands:

```
% su
# inst -f images
# exit
```

5. Run the software. There are two ways to run the software generated by mkrhinoism. To run the software under Irix, kill and restart the toolchest:

```
% killall toolchest
% toolchest
```

The toolchest now has a new menu; with the example inputs above, the name of the new menu would be Volume Manager. The new menu has two items, one for launching a manager program and one for launching a task manager program.

To run the software from a web browser, visit the URL http://*machine*/*Category*Manager/. If mkrhinoism is run on the machine ethyl with the above inputs, the URL to use to access the software would be http://ethyl/VolumeManager/.

P36

60159963 .031500

Putty- 8055

How to use a Rhino NameRenderer

Table of Contents

Introduction

This document is a reference for SGI software engineers who will be using NameRenderers for Rhino applications. NameRenderer is a class that can generate a name that represents a particular Item. Many of the Rhino infrastructure components use a NameRenderer to display the name associated with an Item, including the ItemView, the ItemTable, the ResultView, and the TreeView. The NameRenderer gives the programmer the ability to define what name an Item should have, and have that name used everywhere that the Item's name is displayed.

There is one NameRenderer for each Category. The NameRenderer is responsible for monitoring the Category and generating names for any Items that listeners have expressed interest in.

For more information on NameRenderers in general, see the NameRenderer documentation API documentation.

Before you begin

Before you begin to create a NameRenderer for a particular Category, you need to understand the names and terms that the Rhino infrastructure uses in relation to Categories. See the The names of Categories on the client and on the server documentation for more information.

Specifying the NameRenderer to use for a Category

The HostContext keeps track of which NameRenderer to use for each Category. To specify a Category's NameRenderer, place the NAME_RENDERER resource in the Category's resource file. For example, to specify that the RhinoExampleCategory should use the "com.sgi.rhexamp.category.NameRenderer" class as its NameRenderer, the following entry would be made in

```
/com/sgi/rhexamp/category/rhexampRhinoExampleCategoryP.properties:

com.sgi.rhexamp.category.rhexampRhinoExampleCategory.NameRenderer = com.sgi.rhexamp.
```

If this resource is not specified, then the HostContext creates a ResourceBasedNameRenderer object for the Category.

Using the ResourceBasedNameRenderer

The ResourceBasedNameRenderer is a subclass of NameRenderer that is used to display names i' is no NAME_RENDERER resource specified (see Specifying the NameRenderer to use for a Categor use the ResourceBasedNameRenderer, you must place specific resources in the Category's resource

There are three resources to add to the Category's resource file:

1. The "categoryName" resource, defined as *<category name>.categoryName* (see the CATEGORY documentation for more information). This resource should contain a string that will be used as the name of the Category.
2. The "pluralCategoryName" resource, defined as *<category name>.pluralCategoryName* (see the CATEGORY_PLURAL documentation for more information). This resource should contain a string that will be used as the name of the Category in its plural form.
3. The "nameAttr" resource, defined as *<category name>.nameAttr* (see the NAME documentation for more information). This resource should contain the name of the Item's Attribute whose value will be used as the name of the Item.

As an example of these resource, the RhinoExample Category contains the following entries in its resource file:

```
$(RHINO_EXAMPLE_CATEGORY).categoryName = Rhino Example
$(RHINO_EXAMPLE_CATEGORY).pluralCategoryName = Rhino Examples
$(RHINO_EXAMPLE_CATEGORY).nameAttr = name
```

In this example, the name of the Category will be "Rhino Example", the plural name of the Catego will be "Rhino Examples", and the name of the Item will be the contents of the "name" Attribute Item.

In addition to the three resources listed above, there are three more resources that control the way that the ResourceBasedNameRenderer works. These resources differ from the ones above in that they are not specific to a particular Category. It is expected that this resource will be common to all NameRenderers, and there are default values in the rhino tree. It is possible to override these resources in a product's PackageP.properties file, or individually in a specific Category's resource file.

1. The *ItemAndCategoryFormat* (see the ITEM_NAME_FORMAT documentation for more information). This resource gives the FormatString which will be used to combine the Category name and the Item name. {0} will be replaced by the Item name, {1} by the Category name. This is used in several UI components to display the name of the Item (for example: "Cluster c1").
2. The *ItemView.titleFormatString* (see the IV_TITLE_FORMAT documentation for more

information). This resource gives the FormatString which will be used to combine the name of the Item, the name of the Category, and the name of the server into a single string. The name of the Item will be substituted in {0}, the name of the Category in {1}, and the name of the host that the GUI is connected to in {2}.

3. The *ItemTable.titleFormatString* (see the IT_TITLE_FORMAT documentation for more information). This resource gives the FormatString which will be used to combine the name of the Category and the name of the server into a single string. The name of the Category will be substituted in {0} and the name of the host that the GUI is connected to in {1}.

For example, the default values for these properties, as defined in SysadmUIP.properties, are as follows:

```
A: ItemAndCategoryFormat = {1} {0}
B: ItemView.titleFormatString={1} {0} (on {2})
C: ItemTable.titleFormatString={0} (on {1})
```

Using a subclass of NameRenderer

To provide complete control of the name that is used for an Item, it is also possible to subclass the NameRenderer class and provide the necessary Java code for rendering names. See the NameRenderer documentation for more information. It is also possible to subclass the ResourceBasedNameRenderer to add to the existing functionality of the ResourceBasedNameRenderer.

P38

putty-8056

Welcome to Rhino!

Rhino is an infrastructure for building applications that configure, manage, and monitor hardware and software. Rhino provides a common, consistent, task-based, localized, secure graphical user interface (GUI), with built-in command-line interfaces (CLIs) that system administrators can use to write scripts. Rhino applications consist of two parts:

- **Client-side GUI in Java.** The GUI runs on any platform that has a Java virtual machine, and it doesn't run as root or do setuid root. Rhino provides classes for performing individual tasks, navigating through organized collections of tasks (including a built-in search mechanism), and monitoring the system.
- **Server-side daemon and command-line interfaces.** These are written in C++, so Java doesn't have to run on the server being administered.

Communications between the client and server are secure, non-blocking, and transparent to the application. Encryption is supported (pending export compliance approval), as are security plugins. For a more complete list of features, see Rhino Features.

Your Road to the Code

Each Rhino document begins with this navigation bar containing links to the most important documents:

Welcome | Basic Concepts | GUI Components | Architecture | How To Write An App

Welcome to Rhino is this pleasant and useful overview.

Basic Rhino Concepts is an introduction to concepts which will be used throughout the rest of the documentation. If you'll be building tools with Rhino, start here.

Rhino GUI Components gives screen shots, descriptions, and notes on the most important GUI components provided by Rhino. If you just want to get an idea of what a Rhino application looks like, start here.

Rhino Architecture contains notes and diagrams of the underlying architecture.

How To Write A Rhino Application outlines the steps in writing a Rhino application, and includes links to tutorials.

Throughout your reading, you may find it helpful to refer to the Rhino API documentation: the package index, the class hierarchy, or the index of all fields and methods (big!).

Feedback

Please contact **rhino@newmedia.engr.sgi.com** if you have any comments or questions regarding these documents or the Rhino APIs. Your feedback will help us improve our product and our documentation.

Putty- 8051

Writing Priv Commands

Introduction and Background

As described by the Rhino Architecture document, all communication between the client GUI and the server is handled on the server end by a daemon called sysadmd. This daemon runs as the user who logged into the GUI, not necessarily as root. When a user wants to perform some administrative function that requires root access, a command must be run on the server as root. There are several components of the Rhino architecture which support this:

priv command
: Also known as privileged command. A priv command is a command line program that requires root level access to run successfully. These commands reside in the `/usr/sysadm/privbin` directory, and are the commands that actually perform the changes to the system when the user runs a Task from the GUI. The priv commands are not setuid root, but must usually be run as root to be effective, since they perform operations that will fail if they are not running as root. There are two ways to run priv commands: via the runpriv command, or the root user can invoke them directly.

runpriv
: A setuid program that takes the name of a priv command as an argument. It allows a non-root user to run a priv command as root if any of the following are true:

1. The user is running as root.
2. There is no root password on the system.
3. There is an file in the defaultPrivileges(4) directory granting the privilege to all users.
4. There is an entry in the privilegedUsers(4) database granting the user all privileges.
5. There is an entry in the privilege(4) database granting the user the requested privilege, and the user is not an NIS user.
6. The -auth auth-scheme arguments are provided, and the user passes the authentication test. If auth-scheme is unix, then the user must type the root password when prompted in order to pass.

Privilege Broker Service
: One of the services provided by sysadmd. It allows the GUI to pass a request to run a priv command to the server. The Privilege Broker Service currently always uses the "unix" style of authentication. The GUI can use the `runPriv` family of Java methods (See the Task documentation for more info) to pass commands to the priv broker service.

The Privilege Broker Service is currently the only method for the GUI to run a command on the server. For some products, it may be desirable to have a more general way to run arbitrary commands on the server. These commands would run as the user that logged into the GUI, not necessarily as root, and would therefore have many fewer restrictions. This general "command" service is not currently implemented, but could be written if it is deemed necessary.

Environment of Priv Commands Run Via Runpriv

There are several restrictions placed on priv commands by the runpriv (1M) program for security reasons:

- The priv command must be installed in /usr/sysadm/privbin
- The environment is cleansed of all but the most basic environment variables (see `/var/sysadm/privenviron`)
- The home directory is set to `/var/sysadm/home`
- The priv command runs with the effective uid set to 0 (root) and the real uid set to the uid of the user that logged into sysadmd.

This last restriction makes it impossible to use a script as a priv command, because by default IRIX systems are configured to refuse to run shell scripts if effective uid != real uid. In this case, the recommend solution is to write a C wrapper that sets the uid to the effective uid and then calls the script. For example:

```
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>

#ident "$Revision: 1.2 $"

#define SCRIPT "/usr/bin/script.pl"

/*
 * A C front end to a Perl script.
 * This is required so that runpriv (1M) can
 * execute the script.
 */
void main( int argc, char *argv[])
{

    int   status;
    /* Set the uid to the effective uid
     */
    if (setuid(geteuid()) < 0) {
        perror("setuid");
```

P 40

```
        exit(1);
    }
    status=execv(SCRIPT,argv);
    exit(status);
}
```

Dividing Functionality Among Priv Commands

Depending on the product, it may be desirable to write one priv command that provides all of the functionality necessary, or several priv commands that are specialized. For example, a product that manages user accounts could have one `userAccount` priv command that could add, remove, and modify accounts based on the arguments, or the product could have `addUserAccount`, `removeUserAccount`, and `modifyUserAccount` priv commands.

When deciding on how to break the functionality into priv commands, remember that the runpriv command gives an administrator the ability to grant privileges to users on a *per priv command* basis. Therefore, if you feel that an administrator would benefit from being able to grant privileges to a subset of the functionality of your product, you should divide the priv commands appropriately. For example, in the userAccount case above, if there is only the `userAccount` priv command, then the administrator must grant a user permission to perform all user account functions, or grant no permission. In the case where three priv commands were provided, the administrator could give a particular user permission to add user accounts without giving that user permission to modify or delete existing accounts.

Naming Priv Commands

Because all priv commands reside in a common directory, give priv commands names that will not collide with the priv commands written for other products. It is recommended that you prefix the name of your product's priv commands with a prefix representing the product. For example, `fsmgrAddMachine` would perform the "add machine" functionality for the Failsafe Manager (fsmgr) product.

Returning Status from Priv Commands

While the Rhino infrastructure gives programmers direct access to the return codes and output streams of priv commands, there are some conventions that make things easier:

1. The priv command should return 0 if it was successful. Otherwise return a non-zero error code. This error code can be used by the GUI to present a user-friendly, localized error message, so make sure that the error code is specific enough for the GUI to display a useful message.
2. If the priv command is not successful, then the priv command may send any error output to stderr. By default, the GUI will display this text if the return value is not 0.
3. Any other output that the GUI needs should be sent to stdout.

Validating Input

GUIs written with the Rhino infrastructure often verify that all the arguments to a priv command are valid. This does not mean, however, that the priv command can assume that the arguments are valid. There are several reasons for this:

- The priv command may have been invoked from a shell or script and not from the GUI
- The GUI could have a bug
- The GUI may not be able to validate all inputs
- Due to timing issues, the GUI might not know the correct current state of the system for validation.

For these reasons, it is the responsibility of the priv command to verify that all the inputs are valid before performing any operation.

Priv Commands Should be Atomic

From the user's point of view, a priv command should be an atomic operation that either succeeds completely or fails without making any changes to the system. This is because a half-completed priv command will often leave the system in an inconsistent state that is difficult to diagnose and fix. While this is not always practical, it should be a goal for any priv command.

On a related note, there is nothing to prevent two GUIs from calling the same priv command at the same time. If the system could be corrupted as a result of two or more simultaneous priv commands running, the it is the responsibility of the writer of the priv command to provide some kind of locking mechanism to prevent corruption.

Priv Commands are Logged

All priv commands executed by runpriv are logged in `/var/sysadm/salog`, and `salog` is world-readable. This has several implications:

1. Make sure that no private or secret data is revealed by the priv command on the command line. A way to pass data to a priv command in a secure fashion is discussed below.
2. An user can see a list of all the priv commands that were run. This let users create scripts that call the priv commands simply by cutting and pasting from the log file. If the user is not root, then they will have to use the runpriv command to run the priv commands.
3. The log can often be useful while debugging the GUI and the priv commands to see exactly what commands the GUI ran (or tried to run).

Passing Arguments to Priv Commands

P41

COPY

This section describes the parameter passing conventions used by Rhino applications and the priv commands that they use. The conventions described below are guidelines only - the Rhino infrastructure will allow complete control of the arguments used to start a priv command - but following the conventions will make writing both the GUI Tasks and the priv commands easier because you can take advantage of existing infrastructure for passing the arguments from the client to the server.

The following sections have a lot of detail so that the reader will understand the implications of the way that Rhino passes arguments. Most of the details are taken care of by the infrastructure and libraries supplied with Rhino.

Basic Arguments

Arguments should be passed as key/value pairs, where both the key and value are strings, and the key is separated from the value by a "=" character. The order of the pairs is not important. The following "escapes" are used:

Character	Escape
=	%3d
\n	%0a
%	%25

For example, the priv command to add a user might look like:

```
/usr/sysadm/privbin/addUser username=guest uid=123 homedir=/usr/people/guest "realna
```

This is the way that the client side "runPriv" Java method sends the arguments by default. The key/value style of argument matches the structure of TaskData (the data structure that the Tasks use to store the input collected from users). This makes is so that the Task can automatically convert the data entered by users in a Task to a command line. It also makes the log file easier to understand than the traditional "flag" style of argument specification.

Passing Arrays

There are times when a priv command needs to accept an array of arguments. For example, to add a host to the /etc/hosts file, the priv command might take as many "alias" fields as necessary. In this case, the preferred method is:

1. One key specifies the number of values
2. Each value is passed with a separate key, where the key is formed by appending a number to a prefix.

For example:

```
addHost numAliases=3 alias0=bonnie alias1=bonnie.engr alias2=bonnie.engr.sgi.com
```

This approach obviates the selection of a delimiter character (required in the case of passing an array as, for example, Item=value0,value1,value2) and allows a consistent approach across CLIs.

Passing Args on Stdin

There are some situations where there are too many arguments to fit on the command line, or it's not desirable (perhaps for security reasons) to put a particular argument on the command line. In this case, the priv command can take some arguments on the command line, followed by the special argument "-input". This is a signal to the priv command that it should read the remaining arguments from file descriptor 0 (stdin). The arguments specified on file descriptor 0 are specified as key/value pairs similar to command line arguments, but there are a few differences. Each arguments sent to stdin follows the following format:

1. An 8 character hexadecimal ASCII representation of the number of bytes taken up by the k value, an equal sign, and a newline.
2. A space
3. The key, quoted as described above
4. An equal sign
5. The value, quoted as described above
6. The newline character

The GUI automatically sends any arguments that don't fit on the command line to stdin. It also sends any piece of TaskData that have been marked as hidden (via `setAttrVisible` method of TaskData or AttrBundle) to stdin. The C API (described below) makes reading all of the arguments, both from the command line and stdin, as easy as a few function calls.

C API

To make the writing of priv commands easier, a C API and library are provided that make the reading of arguments a trivial process. The function calls are described first, followed by example code that illustrates how the API is intended to be used.

This is not meant to be a complete description of the SaParam API. See SaParam.h for complete documentation.

Access

Headers for this API are obtained from the header file `sysadm/SaParam.h` (in sysadm_root.sw.hd library that implements the API is `/usr/lib32/libsysadmParam.so.` (in sysadm_root.sw.lib)

Types

The types defined by this API are opaque:

```
# SaParam is a structure that holds all of the parameters, not a single param
typedef struct _SaParam SaParam;
typedef struct _SaParamIter SaParamIter;
```

Create and Destroy

```
extern SaParam *SaParamCreate(void);
```

P42

```
extern void SaParamDestroy(SaParam *param);
```

SaParamCreate() returns NULL if malloc() fails.

Set and Get

```
extern int SaParamSet(SaParam *param, const char *key, const char *value);
extern const char *SaParamGet(SaParam *param, const char *key);
```

SaParamSet() returns 0 if successful, -1 if malloc fails.
SaParamGet() returns NULL if there is no value for "key".

The pointer returned by SaParamGet is owned by "param", and will remain valid as long as SaParamDestroy() or SaParamSet() for this key are not called.

Argument Parsing

```
#define SaPARAM_INPUT_ARG "-input"
extern int SaParamParseArgs(SaParam *param, int argc, char *argv[]);
```

Parse command line arguments of the form "key=value" into key/value pairs. If the argument SaPARAM_INPUT_ARG ("-input") is encountered, read key value pairs from file descriptor 0 as well. Returns 0 if successful, -1 if memory is exhausted, if read fails, or if an unrecognized argument is encountered.

Enumerating the Keys

```
extern SaParamIter *SaParamIterCreate(SaParam *param);
extern void SaParamIterDestroy(SaParamIter *iter);
extern const char *SaParamIterGetKey(SaParamIter *iter);
extern const char *SaParamIterGetValue(SaParamIter *iter);
extern void SaParamIterReset(SaParamIter *iter);
```

To iterate over all of the keys in an SaParam object, create an `SaParamIter` using the `SaParamIterCreate` function, and call `SaParamIterGetKey` repeatedly until it returns NULL. At any point in the iteration, `SaParamIterGetValue` can be called to get the value corresponding to the last key returned by `SaParamIterGetKey`. This is more efficient than calling SaParamGet with each key.

Priv command Sample Code

```
    int main(int argc, char *argv[])
    {
        const char *name = NULL;
        const char *uid = NULL;

        // Create param object
        SaParam param = SaParamCreate();

        // Parse the command line.
        SaParamParseArgs(param, argc, arg);

        name = SaParamGet(param, "name");
        uid = SaParamGet(param, "uid");
        ...
        SaParamDestroy(param);
    }
```

Decoding an array

```
    numParamsString = SaParamGet(params, "numParams");
    if (numParamsString != NULL) {
        numParams = atoi(numParamsString);
        for (i = 0; i < numParams; i++) {
            sprintf(buf, "param%d", i);
            param[i] = SaParamGet(params, buf);
        }
    }
```

Perl API

The following Perl code provides similar functionality as the C API. It is not as complete as the C API, but it is included here for reference. It currently doesn't parse the arguments from stdin, but that functionality can be added if necessary. If this code is useful, it can be added to the Rhino infrastructure.

Priv command Sample Code

```
#!/usr/bin/perl -w

# Parse the command line.
# Pass a reference to @ARGV, and a reference to a hash that will be
# filled in with the arguments
sub parseArgs {

    # Define the "escape" sequences used to quote the command line.
    # These must match those in SaParam.c
    my %escapes = ("3d" => "=", "0a" => "\n", "25" => "%");

    my ($argv, $out) = @_;
    my ($key, $value);
    foreach (@{$argv}) {

        # Split key and value on the "=" character.
        ($key, $value) = split(/=/);

        # Globally replace any sequence of a "%" character
        # followed by two characters with the character from
        # %escapes.  If the sequence is not found in %escapes, then
        # don't replace anything.
        $key   =~ s/%(\w\w)/$escapes{$1}||$&/eg;
        $value =~ s/%(\w\w)/$escapes{$1}||$&/eg;
        $out->{$key} = $value;
    }
}

&parseArgs(\@ARGV, \%args);
foreach $key (sort keys %args) {
    print "$key == $args{$key}\n";
}
```

putty-8058

Rhino ISM Directory Structure

This document describes the directory structure of the ISM (independent software module) generated by the mkrhinoism command.

build

The build directory is for building inst images. build/spec and build/idb describe the packaging of the inst images built by the ISM.

category

The category directory contains subclasses of the Rhino C++ class Category.

cmd

The cmd directory contains the commands used to implement the toolchest menu items for launching the software built by the ISM.

i18n

The i18n directory contains the message catalog for messages printed by server-side components of the ISM.

lib

The lib directory contains library code, which is used to share code among the privileged commands (privcmd) and categories (category) in the ISM.

misc

The misc directory contains the app-chests file that adds a menu to the toolchest for launching the ISM's software.

packages

The packages directory contains all of the Java code in the ISM.

packages/com/sgi/*ism-name*

The packages/com/sgi/*ism-name* directory contains the code for the com.sgi.*ism-name* Java package, which includes the Applet, the TaskManager properties, and constants used by other packages.

packages/com/sgi/*ism-name*/category

The packages/com/sgi/*ism-name*/category directory contains the code for the com.sgi.*ism-name*.category Java package, which contains properties and code for controlling how Items are displayed and for customizing ItemViews and ItemTables.

packages/com/sgi/*ism-name*/ftr

The packages/com/sgi/*ism-name*/ftr directory contains the code for the com.sgi.*ism-name*.ftr package, which contains the FtrIcon subclasses which draw icons for Items in the ISM.

packages/com/sgi/*ism-name*/manager

The packages/com/sgi/*ism-name*/manager directory contains the code for the com.sgi.*ism-name*.manager package, which contains the Manager application for the ISM.

packages/com/sgi/*ism-name*/task

The packages/com/sgi/*ism-name*/task directory contains the code for the com.sgi.*ism-name*.task package, which contains the tasks for the ISM.

privcmd

The privcmd directory contains the privileged commands for the ISM. Privileged commands are run on the server to implement the operations corresponding to tasks.

taskRegistry

The taskRegistry directory specifies the task registry entries for the ISM, which control the lists of tasks in the task manager, in item views, and in result views.

web

The web directory contains the web site for accessing the ISM's software from a browser.

P44

COPY

putty - 8059

Task Internals

This document covers topics of interest to Task writers. See How To Write a Task for the steps in writing a Task.

How Rhino creates and runs a Task

It is important for Task writers to understand the sequence of steps the Rhino infrastructure goes through to create, display, and ultimately perform a Task. This section will briefly describe those steps and the Rhino classes that are involved in the process. For details, please refer to the Rhino API documentation.

Establishing the HostContext

Rhino components need a HostContext in order to access system administration services and share data. The HostContext is created by the login process and typically persists until the last frame of the session has been closed. For example, when the User launches TaskManager, s/he will only have to log in once: any subsequent Task created will share the HostContext created at login time. By contrast, each time the User launches a Task from the command line s/he will have to log in to create a HostContext for the Task.

Creating and loading a Task

Tasks are loaded in a two-step process to allow the Task to be queried for static information such as its User-readable name, icon image, and privileges without having to instantiate the Task. This is especially useful for Task clients that display information about a large number of Tasks, such as TaskManager or TaskShelf, but which don't want the overhead of loading any Task class into memory until it is launched.

The static information about a Task is stored in a text file called a *properties file*, which is similar to an X Windows app-defaults file. The use of properties files for Task writers will be covered in more depth in a later section of this document.

The TaskLoader class is used to implement the two-step loading process. A Task client that wishes to display a Task icon and link but not immediately launch the Task instantiates a TaskLoader, passing in the HostContext and CLASSPATH relative name of the Task. When the User clicks on the link to launch the Task, the client calls TaskLoader.loadTask() to load the Task class and initialize the Task instance.

The TaskLoader.loadTask() method goes through several steps, in order, to initialize a Task and verify that it is ready to run. This sequence is important to Task writers who need to know when each Task method is called during the initialization phase.

1. **Create the TaskContext**
   The TaskContext is used by Task subclasses and their components to share data and state information during the life of a Task. An example of data would be information entered by the User, while state could include information about the server connection.

   One component of TaskContext that is important to Task writers is TaskData. TaskData is a set of key/value pairs, also called *attributes*, representing the information entered by the User as well as other Task state. TaskData can be used to share information among different input components within a Task, as well as among different Tasks in a session. The use of TaskData will be covered in greater detail below.

2. **Load and instantiate the Task class**
   The Task class is loaded into memory and the Task constructor is called.

3. **Load *Product Attributes***
   The optional Product Attributes for a Rhino application is a set of key/value pairs associated with a particular login session. For example, the Product Attributes for the FailSafe product has a single key/value pair "Cluster Name" that specifies the name of the Cluster being managed for a given session. The properties file for a Task specifies what products' Product Attributes must be loaded before the Task will run.

   Product Attributes are stored in the HostContext so that they can be shared by all components in a given session. When the Product Attributes are loaded for the first time, a product-specific plugin is invoked to set the attributes. The plugin may bring up a Frame that requests information from the User. The attribute values are then copied to the TaskData of the Task. Subsequent requests to load Product Attributes will not bring up a Frame, but will simply copy the attribute values cached in the HostContext into the TaskData of the requestor.

4. **Set TaskData attributes**
   Some Task clients may wish to override the Product Attributes or share TaskData attributes among Tasks. For example, a Metatask may wish to pass a TaskData attribute from one Task to the next so that the User doesn't have to enter the data twice. If TaskData attributes are passed to TaskLoader.loadTask(), TaskLoader will attempt to copy those TaskData attributes to the Task being loaded using the Task.setTaskDataAttr() method.

   Not all TaskData attributes may be set by Task clients. Unless a Task has declared an attribute *public* in its properties file, an attempt to call Task.setTaskDataAttr() or Task.getTaskDataAttr() on that attribute will cause the Task to exit with an assertion failure. This mechanism is in place to hide implementaiton details from Task clients. See the Task API documentation for more details about the Task.PUBLIC_DATA property.

5. **Pass operands to the Task**
   Some Tasks may take an operand or operands on which to perform their operation. An operand is typically an Item selector, which is a string that uniquely identifies an administered object on the server. For example, the Modify User Account Task would take a single User Account as an operand, while Delete User Account might take one or more User Accounts as operands and Define User Account would not take any operands. Operands are passed after Product Attributes are loaded and after TaskData atributes are set to allow Product Attributes and TaskData attributes

to be overridden, if desired. TaskLoader will pass operands to the Task being loaded using the method Task.setOperands().

Because operands may be passed to Tasks by a class with no specific knowledge about the Task (for example, a TaskShelf), no ordering of operands should be assumed or required by the Task. See the Task API documentation for more details about operands.

6. **Verify prerequisites**
   The final stage of loading a task is verifying that all of the prerequisites are in place to run the Task. This includes checking the TaskData attributes, operands, privileges, and state of the system being administered.

   The principle behind verifying prerequisites is to detect error conditions as early as possible. For example, a Task that requires special system software to be installed should check the system for that software at this stage of Task loading. It is extremely annoying for Users to enter data and then find out that the system is not in a state to perform the Task.

   TaskLoader calls three different verification methods. This three stage process is again aimed at providing error feedback to the User as early as possible.

   Stage One: Task.verifyPrereqsBeforeCheckPrivs()
   This is the stage where most verification should occur. Only checks that require privileges, such as accessing read-protected files, should be deferred to the third stage.

   Stage Two: Task.checkPrivs()
   Task.checkPrivs() automatically checks the privileges that are defined in the properties file of the Task. If the User does not have the required privileges, s/he will be asked to enter the root password to continue. See the Task API documentation for Task.checkPrivs() and Task.PRIV_LIST for more details.

   Stage Three: Task.verifyPrereqsAfterCheckPrivs()
   This final verification stage is optional, but is provided for those rare Tasks that need privileges to fully verify that the Task is ready to run. For example, Tasks that require access to read-protected files will need to have privileges before being able to verify that the Task prerequisites are met.

Creating the Visible Components of a Task

The visible components of a Task are not created until the Task has been added to a visible Frame or its Frame parent becomes visible for the first time. Even then, visible components are created on a just-in-time basis.

Because Tasks extend the Swing JPanel class, they can be displayed within an existing Frame. However, it is more common to create a new Frame for a Task that has been launched. The TaskFrame class is the canonical container for Tasks. TaskFrame takes care of keeping the Frame title in sync with the state of the Task as well as posting ResultView frames and disposing of the TaskFrame when the Task has been completed or cancelled. A single TaskFrame can even be used to display multiple Tasks sequentially. See the TaskFrame API documentation for more details.

When a Task is made visible for the first time, it calls the method Task.registerInterfaces(), which must be implemented by the Task subclass. registerInterfaces() should create the Form interface and/or Guide interface classes of the Task, and then call Task.setForm() and/or Task.setGuide() to register those interfaces.

At this point there are still no visible components. After the Task subclass has registered its interface(s), the Task must then decide which interface to display. If only one interface has been registered, then the choice is obvious. Otherwise, the property Task.PREFERRED_UI is used to determine which interface should be displayed. See the Task API documentation for more details about the preferred interface.

Next, Task calls either Form.showForm() or Guide.showGuide(), as appropriate.

- When Task calls Form.showForm() for the first time, showForm() calls the abstract method Form.createUI() to initiate creation of the visible components of the Form.

- When Task calls Guide.showGuide() for the first time, showGuide() calls the abstract method Guide.registerPages(). registerPages() should create each of the pages of the guide (but not their visible components) and then call Guide.appendPage() for each page. Finally, showGuide() calls GuidePage.showPage() for the currently active GuidePage, and the visible components for that page are created.

Task writers need only implement the abstract methods of the Task, Form, Guide, and GuidePage classes as described in the API documentation. The complex process of just-in-time creation of visible components described above will happen automatically.

Performing the Task operation

When the User presses "OK", the Task base class first verifies the User data and then tells the subclass to perform the Task operation.

- **Verify User Data**
  One responsibility of the Task subclass constructor is to register a list of TaskDataVerifiers. These verifiers individually check TaskData attributes to confirm that they are valid and together check that the set of attributes is consistent.

  When the User presses "OK", Task calls TaskContext.allDataOK(), which fires each of the registered TaskDataVerifiers in the order they were registered. If a verifier fails, the verification process stops and an error dialog is displayed. After the dialog is dismissed, the User may either press the Cancel button or s/he can modify the data entered and press the OK button again.

- **Initiate Task Operation**
  If the TaskDataVerifiers all succeed, Task calls the abstract Task.ok() method to tell the subclass to initiate the Task operation. The subclass should then invoke one or more privileged commands on the server using a version of Task.runPriv().

  If the Task operation(s) succeed(s), Task.taskSucceeded() must be called. Task.taskSucceeded() notifies TaskDoneListeners that the Task was successful. Typically this leads to the TaskFrame being closed and a ResultViewPanel being displayed. Some versions of Task.runPriv() call

Task.succeeded() automatically, while others require that the Task subclass make the call. See the API documentation for details.

If the Task operation(s) fail(s), Task.taskFailed() must be called. Task.taskFailed() posts an error dialog. After the dialog is dismissed, the User may either press the Cancel button or s/he can modify the data entered and press the OK button again. Some versions of Task.runPriv() call Task.failed() automatically, while others require that the Task subclass make the call. See the API documentation for details.

Other Rhino Classes of Interest to Task Writers

- **ResultListener and ResultEvent**
  The ResultListener and ResultEvent classes are used throughout the Rhino architecture for asynchronous methods. Any method that contacts the server must be asynchronous so that the user interface does not hang during server communication. For example, when a TaskDataVerifier is invoked, a ResultListener is passed to the dataOK() method so that the verifier may contact the server. dataOK() returns immediately to the caller, but the caller should not take any action until one of ResultListener.succeeded() or ResultListener.failed() have been called.

  The ResultListener succeeded() and failed() methods are each passed a ResultEvent object. In general, the result of a successful asynchronous call can be retrieved by ResultEvent.getResult() while the reason for failure of a failed asynchronous call can be retrieved by ResultEvent.getReason(). However, each asynchronous method will have its own protocol for handling success and failure, so check the API documentation for a method before making assumptions.

  When an asynchronous call fails, the ResultListener that initiated the call is typically responsible for posting an error dialog. Sometimes asynchronous calls are chained together, in which case multiple ResultListeners are involved. The general rule is that the ResultListener first in the call chain is responsible for posting an error dialog. For example, the verifyPrereqsBeforeCheckPrivs() method is passed a ResultListener. None of the code within that method or called by that method should be responsible for posting an error dialog. The reason for error should be placed in a ResultEvent object via ResultEvent.setReason() and passed to ResultListener.failed().

- **ResourceStack**
  Properties files in Java traditionally contain text that will be displayed to the User as well as interface characteristics such as fonts, colors and frame sizes. Task properties files contain additional data to support the two-step loading process and reduce the amount of code needed to implement each Task.

  The ResourceStack class was developed to allow the Rhino architecture to supply default property values while providing an opportunity for clients to override those values. The ResourceStack is a stack of ResourceBundles, where a search for a property starts at the top of the stack and works downward until the property is found. Clients can add new properties files to the stack by using the ResourceStack.pushBundle() method. Any property in the most-recently-pushed properties file will override all other properties of the same name lower in the stack. ResourceStack also provides an API that simplifies getting properties of different types, including numeric values, fonts, colors, and String arrays.

- **UIContext**
  A UIContext class is typically shared among related Rhino components to provide access to the ResourceStack and to allow the components to block input or post message dialogs. The UIContext has a *dialog parent* which is the component over which message dialogs should be posted and over which the busy cursor should be displayed when input is blocked. For this reason, UIContext should only be shared among components that are in the same java.awt.Container.

putty-8060

The Rhino TreeViewPane Component

Introduction

The Rhino TreeViewPane is a Component which displays a set of hierarchical data in an outline form. Multiple trees can be displayed, one at a time. The individual nodes in each tree are Items; each level in each tree contains Items within a Category or Association. The structure of the trees are specified almost entirely in Properties Files.

About the TreeViewPane

The TreeViewPane extends the JScrollPane class, and can thus be displayed within any Frame. The tree in the TreeViewPane is a JTree. Each node in the tree is associated with an Item in a particular Category or Association. Each node in the tree has an Icon and a name. The Icon can be a FtrIcon and thus can visually respond to changes in the state of its associated Item. It is possible to customize the display of the name of the Item by specifying an ItemNameRendererFormat.

The JTree in the TreeViewPane is also accessible so that one can take advantage of all of its capabilities (including listening for selections).

Creating a TreeViewPane

As with most Rhino UI classes, there are two basic steps in the creation of a TreeViewPane. First, one adds properties to the properties file which define the structure of the tree. Second, one writes the code which creates a new TreeViewPane which is defined by those properties. The correlation between the properties and the TreeViewPane is a name, a String, which is used as a prefix to the various property names. Pass this string to the TreeViewPane constructor as the *prefix* argument.

The TreeViewPane Properties

The various properties which define the structure of the tree are shown below. Default values, if any, are shown in parentheses.

General Appearance:

`<prefix>.background`
Specifies the color to be used as the background for the TreeViewPane (`#99cccc`).
`<prefix>.width`
Specifies the default width, in points, of the tree pane (`160`).
`<prefix>.height`
Specifies the default height, in points, of the tree pane (`200`).
`<prefix>.toolTipText`
Specifies the string to be displayed as the ToolTip text for each node in the tree.
`<prefix>.textColor`
Specifies the color to be used to display the name of the Item at each node of the tree (`#0033cc`).
`<prefix>.selectColor`
Specifies the color to be used as the background of the selected Item in the tree (`#ffff66`)
`<prefix>.rootFont`
Specifies the name of the font to be used to display the name of the item at the root node of the tree (`SansSerif-bold-12`).
`<prefix>.childFont`
Specifies the name of the font to be used to display the name of all items in the tree (except for the item at the root of the tree) (`SansSerif-12`).
`<prefix>.cellBorderWidth`
Specifies the height, in points, of the border around each item in the tree (`2`).
`<prefix>.cellBorderHeight`
Specifies the width, in points, of the border around each item in the tree (`2`).
`<prefix>.iconWidth`
Specifies the width, in points, of the icon to be displayed at each node of the tree (`17`).
`<prefix>.iconHeight`
Specifies the height, in points, of the icon to be displayed at each node of the tree (`17`).
`<prefix>.iconBlinkOnTime`
Specifies the time, in milliseconds, that a blinking icon will be visible before it blinks off again (`750`).
`<prefix>.iconBlinkOffTime`
Specifies the time, in milliseconds, that a blinking icon will not be visible before it blinks on again (`750`).
`<prefix>.openedIcon`
Specifies the package-qualified name of the icon to display when a node in the tree has children and those children are visible, that is, when the node is open (`com.sgi.sysadm.ftr.OpenArrow`).
`<prefix>.closedIcon`
Specifies the package-qualified name of the icon to display when a node in the tree has children and those children are not visible, that is, when the node is closed (`com.sgi.sysadm.ftr.CloseArrow`).

Tree Structure:

`<prefix>.tree<n>`
A string array that specifies the names of the trees to be displayed in the TreeViewPane. On can be displayed at a time.
`<prefix>.<treename>.level<n>`
The package-qualified name of the Category of Item at each level of the named tree. By default each level of the tree is actually an Association between the Item at the root of the particular subtree and the Category of its children. The Category at the first level of each tree must be the same, and must match the Category of the Item passed to the TreeViewPane constructor.
`<prefix>.<treename>.level<n>.useAssoc`
Specify whether or not to use an Association as the Category for the children of this level (`true`).
`<prefix>.<treename>.level<n>.rootFilterAttr`
If "`.useAssoc`" is false, specify an Attribute of the rootItem. If the value of that Attribute of the root Item of the tree matches the value of that Attribute in each item in the Category, then the item is added to the tree.

Item Rendering:

<prefix>.<categoryName>.displayAttr
Specify this to override the default rendering of the name of each Item in this Category. There are two ways to override the default rendering:

1. Specify an Attribute name; the value of the Attribute will be displayed as the name of the Item (the node); and
2. Specify a format string (see java.text.MessageFormat). The arguments are specified as a .displayAttrArg string array, as below.

<prefix>.<categoryName>.displayAttrArg*<n>*
Each .displayAttrArg (numbered from 0) is an Attribute name. The values of the Item Attributes are passed as arguments to java.text.MessageFormat.

For example, suppose the Properties file contains the following entries:

```
<prefix>.com.shoon.MyCategory.displayAttr = {0}: {1}
<prefix>.com.shoon.MyCategory.displayAttrArg0 = ITEM_TYPE
<prefix>.com.shoon.MyCategory.displayAttrArg1 = ITEM_NAME
```

And let's suppose the Item corresponding to a given node of the tree has the following Attributes:

```
ITEM_TYPE = Personal Name
ITEM_NAME = Howard
```

Then the following call will be made to render the name of the node (using the Attributes of its Item item):

```
java.text.MessageFormat("{0}: {1}",
                        new Object {
                            item.getValueString("ITEM_TYPE"),
                            item.getValueString("ITEM_NAME")
                        });
```

Thus the name of the Item (and the node in the tree) will be rendered as:

```
Personal Name: Howard
```

<prefix>.<categoryName>.stateAttr
The name of the Attribute of the Item to use to determine the state of the Item. The values of this Item Attribute are used to change the rendering of the icon.

<prefix>.<categoryName>.<state>.blink
Set to "true" if the icon should blink when the value of the .stateAttr Attribute of the Item matches "state".

<prefix>.<categoryName>.itemComparator
The fully-qualified name of a Class which is used to compare two Items in this Category. The Class must implement the ItemComparator interface.

Code to Implement a TreeViewPane

To create a new TreeViewPane, you must specify the Item which will serve as the root of the tree and a name which will be used to find the Properties. Note that the type of the Item must match the type of Category specified in the Properties for level0 of the tree. Here is a simple example which creates a TreeViewPane and adds it to the Frame (The *<prefix>* is "MyTree"):

```
TreeViewPane treeViewPane =
    new TreeViewPane(uic, hc, rs, rootItem, "MyTree");
add(treeViewPane);
```

See the description of TreeViewPane for a full description of the Class and its constructor arguments.

The tree displayed by default is tree 0 (see Tree Structure above).

To change trees programmatically, for example, using a menu, call TreeViewPane.setTree(int) or TreeViewPane.setTree(java.lang.String).

To listen for user selection of a node in the tree, use the standard JTree calls. For example, use TreeViewPane.addTreeSelectionListener(TreeSelectionListener) to add a listener which fires when a node in the tree is selected. You can also use TreeViewPane.addActionListener(ActionListener) to listen for the user performing an action upon a node in the tree.

Examples:

Here is a portion of the Properties file which defines the structure of four tree (example adapted from the FailSafe Manager 2.0 product):

```
MyTree.tree0 = groupsResources
MyTree.tree1 = resources
MyTree.tree2 = groups
MyTree.tree3 = policies

MyTree.groupsResources.level0 = com.sgi.fsmgr.category.ClusterCategory
MyTree.groupsResources.level1 = com.sgi.fsmgr.category.ResourceGroupCategory
MyTree.groupsResources.level2 = com.sgi.fsmgr.category.ResourceCategory

MyTree.resources.level0 = com.sgi.fsmgr.category.ClusterCategory
MyTree.resources.level1 = com.sgi.fsmgr.category.ResourceCategory

MyTree.groups.level0 = com.sgi.fsmgr.category.ClusterCategory
MyTree.groups.level1 = com.sgi.fsmgr.category.ResourceGroupCategory

MyTree.policies.level0 = com.sgi.fsmgr.category.ClusterCategory
MyTree.policies.level1 = com.sgi.fsmgr.category.FailoverPolicyCategory
MyTree.policies.level1.useAssoc = false

MyTree.com.sgi.fsmgr.category.ResourceCategory.displayAttr = {0}: {1}
MyTree.com.sgi.fsmgr.category.ResourceCategory.displayAttrArg0 = _RESOURCE_TYP
MyTree.com.sgi.fsmgr.category.ResourceCategory.displayAttrArg1 = _RESOURCE
MyTree.com.sgi.fsmgr.category.ResourceCategory.stateAttr = CAM_STATUS
MyTree.com.sgi.fsmgr.category.ResourceCategory.ONLINE_PENDING.blink = true
MyTree.com.sgi.fsmgr.category.ResourceCategory.OFFLINE_PENDING.blink = true
MyTree.com.sgi.fsmgr.category.ResourceCategory.ERROR.blink = true
MyTree.com.sgi.fsmgr.category.ResourceCategory.itemCompare = \
        com.sgi.fsmgr.detailView.CategoryItemCompare

MyTree.com.sgi.fsmgr.category.ResourceGroupCategory.stateAttr = CAM_STATUS
MyTree.com.sgi.fsmgr.category.ResourceGroupCategory.ERROR.blink = true
MyTree.com.sgi.fsmgr.category.ResourceGroupCategory.ONLINE_PENDING.blink = tru
```

P49

```
MyTree.com.sgi.fsmgr.category.ResourceGroupCategory.OFFLINE_PENDING.blink = tr
MyTree.com.sgi.fsmgr.category.ResourceGroupCategory.itemCompare = \
        com.sgi.fsmgr.detailView.CategoryItemCompare

MyTree.com.sgi.fsmgr.category.ClusterCategory.stateAttr = CAM_STATUS
MyTree.com.sgi.fsmgr.category.ClusterCategory.INACTIVE.blink = true
```

These example Properties define four (4) trees, one of which is displayed in the TreeViewPane at any given time. The Item at the root of the tree must be in the "`Cluster`" Category. Any given tree can be dynamically chosen for display. To select the "`resources`" tree, for example, to be displayed in the TreeViewPane, the following calls are equivalent:

```
treeViewPane.setTree(1);
treeViewPane.setTree("resources");
```

The four trees which can be displayed are as follows:

1. `groupsResources`
   This tree is three (3) levels deep. The second level of the tree is populated with Items in an Association between the root `Cluster` Item and Items in the "`ResourceGroup`" Category. The third level of the tree is populated with Items in an Association between each `ResourceGroup` Item at the second level of the tree and Items in the "`Resource`" Category.
2. `resources`
   This tree is two (2) levels deep. The second level of the tree is populated with Items in an Association between the root `Cluster` Item and Items in the "`Resource`" Category.
3. `groups`
   This tree is two (2) levels deep. The second level of the tree is populated with Items in an Association between the root `Cluster` Item and Items in the "`ResourceGroup`" Category.
4. `policies`
   This tree is two (2) levels deep. The second level of the tree is populated with Items in the "`FailoverPolicy`" Category (no Association is used).

Four (4) Categories of Items can be displayed in the tree, as follows:

1. `com.sgi.fsmgr.category.ClusterCategory`
   If the "`CAM_STATUS`" Attribute of any Item in this Category has the value "`INACTIVE`", its icon will blink. Names of Items in this Category will be rendered using the default rendering.
2. `com.sgi.fsmgr.category.ResourceGroupCategory`
   If the "`CAM_STATUS`" Attribute of any Item in this Category has the value "`ONLINE_PENDING`" "`ERROR`", or "`OFFLINE_PENDING`", its icon will blink. Items in this Category will be compared (for sorting purposes) by using the ItemComparator
   `com.sgi.fsmgr.detailView.CategoryItemCompare`. Names of Items in this Category will be rendered using the default rendering.
3. `com.sgi.fsmgr.category.ResourceCategory`
   If the "`CAM_STATUS`" Attribute of any Item in this Category has the value "`ONLINE_PENDING`" "`ERROR`", or "`OFFLINE_PENDING`", its icon will blink. The name of the Item will be rendered using the java.text.MessageFormat string "`{0}: {1}`" with the arguments being the "`_RESOURCE_TYPE`" and "`_RESOURCE`" Attributes of that Item, respectively. Items in this Category will be compared (for sorting purposes) by using the ItemComparator
   `com.sgi.fsmgr.detailView.CategoryItemCompare`.
4. `com.sgi.fsmgr.category.FailoverPolicyCategory`
   Names of Items in this Category will be rendered using the default rendering.

To listen for a node in the tree being acted upon (double-clicked) by the user, the following code is used:

```
treeViewPane.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {

        // get the node of the tree that has been selected
        //
        DefaultMutableTreeNode node = (DefaultMutableTreeNode)
            (((TreePath)event.getSource()).getLastPathComponent());

        // ... node actions go here ...

        try {

            // get the Item that belongs to the selected node
            //
            ItemUserObject nodeInfo
                = (ItemUserObject)node.getUserObject();
            Item item = nodeInfo.getItem();

            // ... actions upon the Item go here ...

        } catch (ClassCastException ex) {
        }
    }
});
```

To listen for a node in the tree being selected by the user, the following code is used:

```
treeViewPane.addTreeSelectionListener(new TreeSelectionListener() {
    public void valueChanged(TreeSelectionEvent event) {

        // get the node of the tree that has been selected
        //
        DefaultMutableTreeNode node = (DefaultMutableTreeNode)
            (event.getPath().getLastPathComponent());

        // ... node actions go here ...

        try {

            // get the Item that belongs to the selected node
            //
            ItemUserObject nodeInfo
                = (ItemUserObject)node.getUserObject();
            Item item = nodeInfo.getItem();

            // ... actions upon the Item go here ...

        } catch (ClassCastException ex) {
        }
    }
});
```

Class Hierarchy

- class java.lang.Object
  - class com.sun.java.swing.table.AbstractTableModel (implements com.sun.java.swing.table.TableModel, java.io.Serializable)
    - class com.sgi.sysadm.ui.ItemTableModel
  - class com.sgi.sysadm.ui.HostContext.AppExitHandler (implements com.sgi.sysadm.ui.HostContext.ExitHandler)
  - class com.sgi.sysadm.ui.HostContext.AppletHelpProvider (implements com.sgi.sysadm.ui.HostContext.HelpProvider)
  - class com.sgi.sysadm.ui.ArrowIcon (implements com.sun.java.swing.Icon)
  - class com.sgi.sysadm.util.AttrBundle
    - class com.sgi.sysadm.category.Category
      - class com.sgi.sysadm.category.Association
    - class com.sgi.sysadm.category.Item
    - class com.sgi.sysadm.ui.taskData.TaskData
  - interface com.sgi.sysadm.util.AttrListener (extends java.util.EventListener)
  - class com.sgi.sysadm.util.Attribute
  - interface com.sgi.sysadm.util.BinaryPredicate
  - class com.sun.java.swing.ButtonGroup (implements java.io.Serializable)
    - class com.sgi.sysadm.ui.RButtonGroup
  - class com.sgi.sysadm.category.CategoryAdapter (implements com.sgi.sysadm.category.CategoryListener)
  - interface com.sgi.sysadm.category.CategoryListener (extends com.sgi.sysadm.util.AttrListener)
  - class com.sgi.sysadm.ui.EditableList.Cell
  - class com.sgi.sysadm.ui.HostContext.Client
  - class java.awt.Component (implements java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable)
    - class java.awt.Container
      - class com.sun.java.swing.JComponent (implements java.io.Serializable)
        - class com.sun.java.swing.AbstractButton (implements java.awt.ItemSelectable, com.sun.java.swing.SwingConstants)
          - class com.sun.java.swing.JToggleButton (implements com.sun.java.accessibility.Accessible)
            - class com.sun.java.swing.JCheckBox (implements com.sun.java.accessibility.Accessible)
              - class com.sgi.sysadm.ui.RCheckBox (implements com.sgi.sysadm.ui.ExtraCleanup)
            - class com.sun.java.swing.JRadioButton (implements com.sun.java.accessibility.Accessible)
              - class com.sgi.sysadm.ui.RRadioButton (implements com.sgi.sysadm.ui.ExtraCleanup)
        - class com.sun.java.swing.JComboBox (implements java.awt.ItemSelectable, com.sun.java.swing.event.ListDataListener, java.awt.event.ActionListener, com.sun.java.accessibility.Accessible)
          - class com.sgi.sysadm.ui.ItemFinder (implements com.sgi.sysadm.ui.ExtraCleanup)
        - class com.sun.java.swing.JLabel (implements com.sun.java.swing.SwingConstants, com.sun.java.accessibility.Accessible)
          - class com.sgi.sysadm.ui.RLabel
            - class com.sgi.sysadm.ui.ValueLabel
        - class com.sun.java.swing.JPanel (implements com.sun.java.accessibility.Accessible)
          - class com.sgi.sysadm.ui.EditableList
          - class com.sgi.sysadm.ui.Guide
          - class com.sgi.sysadm.ui.ItemTablePanel (implements com.sgi.sysadm.ui.ExtraCleanup, com.sgi.sysadm.ui.ItemTableProperties)
          - class com.sgi.sysadm.ui.ItemViewPanel (implements com.sgi.sysadm.ui.ItemViewProperties)
          - class com.sgi.sysadm.manager.taskManager.MetataskGroupPanel (implements com.sgi.sysadm.manager.taskManager.TaskManagerProperties)
          - class com.sgi.sysadm.ui.RPanel (implements com.sgi.sysadm.ui.DynamicSize)
            - class com.sgi.sysadm.ui.LabelComponentPanel
            - class com.sgi.sysadm.ui.OneColumnPanel
              - class com.sgi.sysadm.ui.TaskLaunchComponent
            - class com.sgi.sysadm.ui.TaskPage
              - class com.sgi.sysadm.ui.Form
              - class com.sgi.sysadm.ui.GuidePage
            - class com.sgi.sysadm.ui.TwoColumnPanel
          - class com.sgi.sysadm.ui.ResultViewPanel
          - class com.sgi.sysadm.manager.taskManager.TableOfContents (implements com.sgi.sysadm.manager.taskManager.TaskManagerProperti
          - class com.sgi.sysadm.ui.Task
          - class com.sgi.sysadm.ui.TaskControlPanel
          - class com.sgi.sysadm.ui.TaskShelfPanel
        - class com.sun.java.swing.JScrollPane (implements com.sun.java.swing.ScrollPaneConstants, com.sun.java.accessibility.Accessible)
          - class com.sgi.sysadm.manager.taskManager.DisplayArea (implements com.sgi.sysadm.manager.taskManager.TaskManagerProperties)
          - class com.sgi.sysadm.ui.treeView.TreeViewPane (implements com.sgi.sysadm.ui.treeView.TreeViewProperties, com.sgi.sysadm.ui.ExtraCleanup)
        - class com.sun.java.swing.JTable (implements com.sun.java.swing.event.TableModelListener,

P51

COPY

com.sun.java.swing.Scrollable,
com.sun.java.swing.event.TableColumnModelListener,
com.sun.java.swing.event.ListSelectionListener,
com.sun.java.swing.event.CellEditorListener,
com.sun.java.accessibility.Accessible)

- ■ class com.sgi.sysadm.ui.ComponentTable (implements java.awt.event.MouseListener, java.awt.event.MouseMotionListener)

■ class com.sun.java.swing.text.JTextComponent (implements com.sun.java.swing.Scrollable, com.sun.java accessibility.Accessible)
- ■ class com.sun.java.swing.JTextField (implements com.sun.java.swing.SwingConstants)
  - ■ class com.sun.java.swing.JPasswordField
    - ■ class com.sgi.sysadm.ui.RPasswordField
  - ■ class com.sgi.sysadm.ui.RTextField (implements com.sgi.sysadm.ui.ExtraCleanup)
    - ■ class com.sgi.sysadm.ui.FilteredTextField
    - ■ class com.sgi.sysadm.ui.IntegerTextField

■ class com.sgi.sysadm.ui.richText.RichTextComponent (implements com.sgi.sysadm.ui.DynamicSize, com.sgi.sysadm.ui.DynamicCursor)
- ■ class com.sgi.sysadm.ui.richText.RichTextArea
  - ■ class com.sgi.sysadm.ui.LinkLabelBase
    - ■ class com.sgi.sysadm.ui.LinkLabel
    - ■ class com.sgi.sysadm.ui.OptionalLinkLabel

■ class java.awt.Panel
- ■ class java.applet.Applet
  - ■ class com.sun.java.swing.JApplet (implements com.sun.java.accessibility.Accessible, com.sun.java.swing.RootPaneContainer)
    - ■ class com.sgi.sysadm.manager.RApplet

■ class java.awt.Window
- ■ class java.awt.Dialog
  - ■ class com.sun.java.swing.JDialog (implements com.sun.java.swing.WindowConstants, com.sun.java.accessibility.Accessible, com.sun.java.swing.RootPaneContainer)
    - ■ class com.sgi.sysadm.ui.RDialog
- ■ class java.awt.Frame (implements java.awt.MenuContainer)
  - ■ class com.sun.java.swing.JFrame (implements com.sun.java.swing.WindowConstants, com.sun.java.accessibility.Accessible, com.sun.java.swing.RootPaneContainer)
    - ■ class com.sgi.sysadm.ui.RFrame
      - ■ class com.sgi.sysadm.ui.ItemTableFrame (implements com.sgi.sysadm.ui.ItemTableProperties)
      - ■ class com.sgi.sysadm.ui.ItemViewFrame (implements com.sgi.sysadm.ui.ItemViewProperties)
      - ■ class com.sgi.sysadm.plugin.LogViewerFrame
      - ■ class com.sgi.sysadm.ui.TaskFrame (implements

com.sgi.sysadm.ui.event.TitleListener,
com.sgi.sysadm.ui.event.TaskDoneListener)

- ■ class com.sgi.sysadm.ui.ResultViewFrame

○ class com.sgi.sysadm.ui.EditableList.DefaultEditVerifier (implements com.sgi.sysadm.ui.EditableList.EditVerifier)
○ class com.sgi.sysadm.manager.taskManager.DisplayPage (implements com.sgi.sysadm.manager.taskManager.TaskManagerProperties)
○ interface com.sgi.sysadm.ui.DynamicCursor
○ interface com.sgi.sysadm.ui.DynamicSize
○ interface com.sgi.sysadm.ui.DynamicSizeLayoutManager
○ interface com.sgi.sysadm.ui.EditableList.EditVerifier
○ class java.util.EventObject (implements java.io.Serializable)
- ■ class com.sgi.sysadm.util.AttrEvent
- ■ class com.sgi.sysadm.ui.event.GuideNavigationEvent
- ■ class com.sgi.sysadm.ui.event.ItemFinderEvent
- ■ class com.sgi.sysadm.ui.event.ItemTableLaunchRequestEvent
- ■ class com.sgi.sysadm.ui.event.ItemViewLaunchRequestEvent
- ■ class com.sgi.sysadm.ui.richText.LinkEvent
- ■ class com.sgi.sysadm.util.ProcessEvent
- ■ class com.sgi.sysadm.manager.RApp.RAppLaunchEvent
- ■ class com.sgi.sysadm.util.ResultEvent
- ■ class com.sgi.sysadm.ui.event.TableSortRequestEvent
- ■ class com.sgi.sysadm.ui.taskData.TaskDataEvent
- ■ class com.sgi.sysadm.ui.event.TaskLaunchComponentEvent
- ■ class com.sgi.sysadm.ui.event.TaskLaunchRequestEvent
- ■ class com.sgi.sysadm.ui.event.TaskResult
- ■ class com.sgi.sysadm.ui.event.TitleEvent

○ interface com.sgi.sysadm.ui.HostContext.ExitHandler
○ interface com.sgi.sysadm.ui.ExtraCleanup
○ class com.sgi.sysadm.ui.FtrIcon (implements com.sun.java.swing.Icon)
○ class com.sgi.sysadm.ui.GenericItemRenderer
- ■ class com.sgi.sysadm.ui.IconRenderer
  - ■ class com.sgi.sysadm.ui.ResourceBasedIconRenderer
- ■ class com.sgi.sysadm.ui.NameRenderer
  - ■ class com.sgi.sysadm.ui.ResourceBasedNameRenderer

○ interface com.sgi.sysadm.ui.HostContext.HelpProvider
○ class com.sgi.sysadm.ui.HostContext
○ interface com.sgi.sysadm.category.ItemComparator
○ interface com.sgi.sysadm.ui.ItemDisplayNameRenderer
○ interface com.sgi.sysadm.ui.event.ItemFinderListener (extends java.util.EventListener)
○ class com.sgi.sysadm.ui.ItemFinderState
○ class com.sgi.sysadm.ui.ItemNameRendererFormat
○ class com.sgi.sysadm.ui.ItemTable (implements java.awt.ItemSelectable, com.sgi.sysadm.ui.ItemTableProperties)
○ class com.sgi.sysadm.ui.ItemTableColumn
○ interface com.sgi.sysadm.ui.ItemTableColumnRenderer
○ class com.sgi.sysadm.ui.ItemTableController (implements com.sgi.sysadm.ui.ItemTableProperties)

P 52

- class com.sgi.sysadm.ui.ItemTableFactory (implements com.sgi.sysadm.ui.ItemTableProperties)
- interface com.sgi.sysadm.ui.event.ItemTableLaunchRequestListener (extends java.util.EventListener)
- interface com.sgi.sysadm.ui.ItemTableProperties
- class com.sgi.sysadm.category.ItemTestResult
- interface com.sgi.sysadm.category.ItemTester
- class com.sgi.sysadm.ui.treeView.ItemUserObject (implements com.sgi.sysadm.ui.treeView.TreeViewProperties, com.sgi.sysadm.ui.ExtraCleanup)
- class com.sgi.sysadm.ui.ItemView (implements com.sgi sysadm.ui.ItemViewProperties)
- interface com.sgi.sysadm.ui.ItemViewAdditionalInfoRenderer
- class com.sgi.sysadm.ui.ItemViewController (implements com.sgi.sysadm.ui.ItemViewProperties)
- class com.sgi.sysadm.ui.ItemViewFactory (implements com.sgi.sysadm.ui.ItemViewProperties)
- interface com.sgi.sysadm.ui.ItemViewFieldRenderer
- class com.sgi.sysadm.ui.ItemViewInfo
- interface com.sgi.sysadm.ui.event.ItemViewLaunchRequestListener (extends java.util.EventListener)
- interface com.sgi.sysadm.ui.ItemViewProperties
- class com.sgi.sysadm.ui.LabelComponentConstraints (implements java.lang.Cloneable)
- interface com.sgi.sysadm.ui.richText.LinkListener (extends java.util.EventListener)
- class com.sgi.sysadm.ui.LinkPageLayout (implements java.awt.LayoutManager2)
- class com.sgi.sysadm.util.Log
- class com.sgi.sysadm.plugin.LogViewer (implements com.sgi.sysadm.manager.taskManager.TaskManagerFrame)
- class com.sgi.sysadm.category.NotificationFilter
- interface com.sgi.sysadm.util.PrivBroker
- interface com.sgi.sysadm.util.ProcessListener
- class com.sgi.sysadm.util.ProcessWatcher
- interface com.sgi.sysadm.ui.ProductAttributeSetter
- class com.sgi.sysadm.manager.RApp
  - class com.sgi.sysadm.manager.RunItemTable
  - class com.sgi.sysadm.manager.RunItemView
  - class com.sgi.sysadm.manager.RunTask
  - class com.sgi.sysadm.manager.TaskManager (implements com.sgi.sysadm.manager.taskManager.TaskManagerProperties)
- interface com.sgi.sysadm.manager.RApp.RAppLaunchListener (extends java.util.EventListener)
- interface com.sgi.sysadm.ui.RButtonGroup.RButtonGroupListener (extends java.util.EventListener)
- interface com.sgi.sysadm.ui.RenderedObjectListener
- class com.sgi.sysadm.util.ResourceStack (implements java.lang.Cloneable)
- class com.sgi.sysadm.util.ResultAdapter (implements com.sgi.sysadm.util.ResultListener)
- interface com.sgi.sysadm.util.ResultListener (extends java.util.EventListener)
- class com.sgi.sysadm.ui.HostContext.SGIHelpProvider (implements com.sgi.sysadm.ui.HostContext.HelpProvider)
- class com.sgi.sysadm.manager.taskManager.SearchPanel (implements com.sgi.sysadm.manager.taskManager.TaskManagerPanel, com.sgi.sysadm.manager.taskManager.TaskManagerProperties)
- class com.sgi.sysadm.util.Sort
- class com.sgi.sysadm.util.SysUtil
- class com.sgi.sysadm.ui.TableCellComponentRenderer (implements com.sun.java.swing.table.TableCellRenderer)
- interface com.sgi.sysadm.ui.event.TableSortRequestListener (extends java.util.EventListener)
- class com.sgi.sysadm.ui.event.TaskControlAdapter (implements com.sgi.sysadm.ui.event.TaskControlListener)
- interface com.sgi.sysadm.ui.event.TaskControlListener (extends java.util.EventListe
- class com.sgi.sysadm.ui.taskData.TaskDataBinder (implements com.sgi.sysadm.util.AttrListener)
  - class com.sgi.sysadm.ui.taskData.AssociationItemFinderBinder
  - class com.sgi.sysadm.ui.taskData.AttributeAbstractButtonBinder (implements com.sun.java.swing.event.ChangeListener)
  - class com.sgi.sysadm.ui.taskData.BooleanAbstractButtonBinder (implements com.sun.java.swing.event.ChangeListener)
  - class com.sgi.sysadm.ui.taskData.BooleanComponentEnabledBinder
  - class com.sgi.sysadm.ui.taskData.LongJComboBoxBinder
  - class com.sgi.sysadm.ui.taskData.LongJTextComponentBinder (implements com.sun.java.swing.event.DocumentListener)
  - class com.sgi.sysadm.ui.taskData.LongRButtonGroupBinder
  - class com.sgi.sysadm.ui.taskData.ReasonItemFinderBinder (implements com.sgi.sysadm.ui.event.ItemFinderListener)
  - class com.sgi.sysadm.ui.taskData.SelectorItemFinderBinder (implements com.sgi.sysadm.ui.event.ItemFinderListener)
  - class com.sgi.sysadm.ui.taskData.StringComponentEnabledBinder
  - class com.sgi.sysadm.ui.taskData.StringJComboBoxBinder
  - class com.sgi.sysadm.ui.taskData.StringJLabelBinder
  - class com.sgi.sysadm.ui.taskData.StringJTextComponentBinder (implements com.sun.java.swing.event.DocumentListener)
  - class com.sgi.sysadm.ui.taskData.StringTaskLaunchComponentBinder
  - class com.sgi.sysadm.ui.taskData.TextItemFinderBinder (implements com.sgi.sysadm.ui.event.ItemFinderListener)
- interface com.sgi.sysadm.ui.taskData.TaskDataVerifier
- interface com.sgi.sysadm.ui.event.TaskDoneListener (extends java.util.EventListener)
- interface com.sgi.sysadm.ui.event.TaskLaunchComponentListener (extends java.util.EventListener)
- class com.sgi.sysadm.ui.TaskLaunchComponentState
- interface com.sgi.sysadm.ui.event.TaskLaunchRequestListener (extends java.util.EventListener)
- class com.sgi.sysadm.ui.TaskLoader
- interface com.sgi.sysadm.manager.taskManager.TaskManagerAction (extends com.sgi.sysadm.manager.taskManager.TaskManagerPlugin)
- interface com.sgi.sysadm.manager.taskManager.TaskManagerFrame (extends com.sgi.sysadm.manager.taskManager.TaskManagerPlugin)
- interface com.sgi.sysadm.manager.taskManager.TaskManagerInitPlugin (extends

P53

com.sgi.sysadm.manager.taskManager.TaskManagerPlugin)
- class com.sgi.sysadm.manager.taskManager.TaskManagerLinkListener (implements com.sgi.sysadm.ui.richText.LinkListener, com.sgi.sysadm.manager.taskManager.TaskManagerProperties)
- interface com.sgi.sysadm.manager.taskManager.TaskManagerPanel (extends com.sgi.sysadm.manager.taskManager.TaskManagerPlugin)
- interface com.sgi.sysadm.manager.taskManager.TaskManagerPlugin
- interface com.sgi.sysadm.manager.taskManager.TaskManagerProperties
- interface com.sgi.sysadm.manager.taskManager.TaskManagerTitleRenderer (extends com.sgi.sysadm.manager.taskManager.TaskManagerPlugin)
- class com.sgi.sysadm.manager.taskManager.TaskManagerUtil (implements com.sgi.sysadm.manager.taskManager.TaskManagerProperties)
- class com.sgi.sysadm.ui.TaskRegistry
- class com.sgi.sysadm.ui.TaskShelfController
- class java.lang.Throwable (implements java.io.Serializable)
    - class java.lang.Exception
        - class com.sgi.sysadm.util.SysUtil.ClassLoadException
        - class com.sgi.sysadm.ui.ItemTableException
        - class com.sgi.sysadm.ui.ItemViewException
        - class com.sgi.sysadm.ui.TaskInitFailedException
        - class com.sgi.sysadm.ui.TaskLoaderException
- interface com.sgi.sysadm.ui.event.TitleListener (extends java.util.EventListener)
- interface com.sgi.sysadm.ui.treeView.TreeViewProperties
- class com.sgi.sysadm.ui.UIContext
    - class com.sgi.sysadm.ui.ItemTableContext
    - class com.sgi.sysadm.ui.ItemViewContext
    - class com.sgi.sysadm.ui.TaskContext

p54

Rhino Classes

Rhino classes provide a means to represent the system to be administered and to perform user-specified operations to change system state:

- **Item** represents some system entity to be administered (ex., a cluster, an XLV volume, a filesystem). An Item usually has a unique icon in the GUI, reflecting its current state.
- **ItemView** is a window displaying relevant information about an Item.
- **Category** is a class of Items (ex., cluster nodes).
- **CategoryView** is a window displaying relevant information about a Category.
- **TreeView** displays Items that have a natural hierarchical relationship in an outline-style indented overview. Because it lets the user monitor the states of several Items at once, TreeView can be appropriate in a front-end monitoring window.
- **Task** is a window that users interact with. When the user clicks the OK button, the Task calls a CLI on the server to perform an atomic operation that changes the state of an Item. Tasks can be combined to achieve a high-level goal (a "metatask").
- **ResultView** is a window that appears when the user has completed a Task successfully, presenting a list of Tasks that the user may want to launch next.
- **Task Manager** is a front-end window that organizes and lists all of a product's Tasks, for easy access.

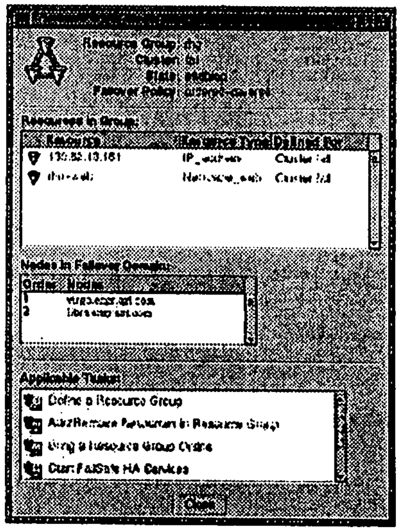
For all applications that the Rhino team has encountered, the above elements sufficed to satisfy administration requirements. If you encounter a requirement that is not satisfied by any of the above components, please email the Rhino team. We await your feedback for additional requirements.

Screenshot Examples

The following screenshots have been scaled to 60% their actual size. These examples are taken from the FailSafe 2.0 GUI product which is based on Rhino.

ItemView

The ItemView window displays simple key-value pairs at the top, application-specific contents in ItemTables in the middle, and a task shelf at the bottom. The Item's icon is shown at the top left, with the icon color indicating the Item's state.



Resource Group: rho
Cluster: fall
State: [illegible]
Failover Policy: [illegible]

Resources in Group:

Resource	Resource Type	Defined For
13?.5?.13.161	IP_address	Cluster fall
[illegible]	Netscape_web	Cluster fall

Nodes in Failover Domain:

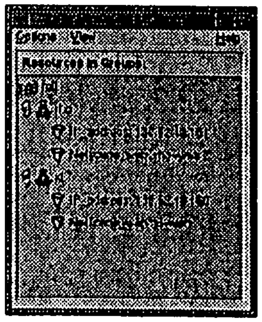
Order	Node
1	[illegible]
2	[illegible]

Applicable Tasks:

- Define a Resource Group
- Add/Remove Resources in Resource Group
- Bring a Resource Group Online
- Start FailSafe HA Services

TreeView

The TreeView shows Items that have a hierarchical relationship. In this example from FailSafe 2.0 GUI, three different kinds of Items are shown; the cluster "fall" contains two resource groups "rho" and "xi," and each resource group contains two resources.



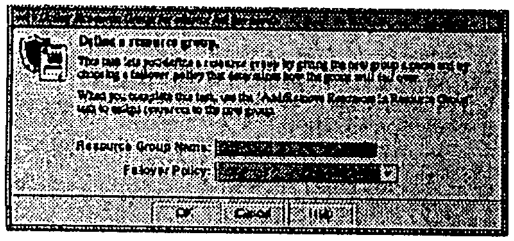
P55

Task

The Task window has a product-specific Task icon in the upper left corner (in this case, the FailSafe 2.0

COPY

GUI shield logo behind the generic Rhino Task logo). After the Task title, some introText follows, describing the inputs the user is expected to type or choose. Application-specific inputs themselves appear at the bottom. All text in blue behaves like a hyperlink and launches glossary information in a separate small window.



Define a resource group.

This task lets you define a resource group by giving the new group a name and by choosing a failover policy that determines how the group will fail over.

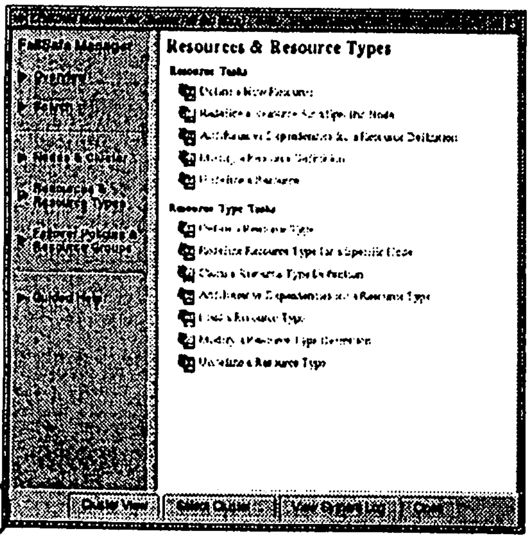
When you complete this task, use the "Add/Remove Resources in Resource Group" task to assign resources to the new group.

Resource Group Name:

Failover Policy:

Task Manager

The Task Manager groups the product's Tasks into pages based on the different types of Items that the Tasks operate upon. The pages appear on the left side in the table of contents. The Overview and Search pages appear in all Rhino applications, but the text content on the Overview page is application-specific (describes the application and the application-specific categories). For the FailSafe 2.0 GUI, all metatasks are grouped into a sixth page called Guided Help, rather than presented in a special section at the top of each page.



FailSafe Manager
Overview
Search
Nodes & Cluster
Resources & Resource Types
Failover Policies & Resource Groups
Guided Help

Resources & Resource Types

Resource Tasks

- Define a New Resource
- Redefine a Resource for a Specific Node
- Add/Remove Dependencies for a Resource Definition
- Modify a Resource Definition
- Undefine a Resource

Resource Type Tasks

- Define a Resource Type
- Redefine a Resource Type for a Specific Node
- Clone a Resource Type Definition
- Add/Remove Dependencies for a Resource Type
- Load a Resource Type
- Modify a Resource Type Definition
- Undefine a Resource Type

p56

How to Write a Rhino Application

The following steps are meant to be broad guidelines; your particular application may require more or fewer steps than those listed below.

1. **Analyze the UI requirements of your application** in terms of the Rhino UI classes. Understand the Rhino users model.
   - GUI Design and Implementation
     - Analyzing UI Requirements
   - GUI Components
   - Rhino Users Model

2. **Analyze server-side requirements.** Identify the Category(s) that need to be administered. Identify Item attributes and Category attributes (if any).

3. **Generate example ISM.** Use mkrhinoism to generate a self-building example ISM that will help you get started using Rhino to create your application.

4. **Provide an API** that allows Rhino to determine the Item(s) belonging to the Category(s) and provide notification of changes to the state of the Item(s).

5. **Implement the server-side functionality:**
   - Priv Commands
   - Item and Category
   - Association

6. **Implement the client-side user interfaces:**
   - Item and Category
   - Associations
   - Tasks
   - Icon Renderers
   - Name Renderers
   - ItemViews
   - ItemTables
   - TreeViewPanes
   - Task Manager

P57

Rhino GUI Components

Rhino provides some high-level GUI elements for displaying Tasks, collections of tasks, the status and relationships of Items operated on by those Tasks, and the results of those Tasks.

The following screen shots have been scaled to 50% of their actual size, and are links to full-size images. These examples are taken from the FailSafe 2.0 Cluster Manager GUI.

Windows:

- Form
- Guide
- ResultView
- TaskManager
- MetaTask
- Smart MetaTask

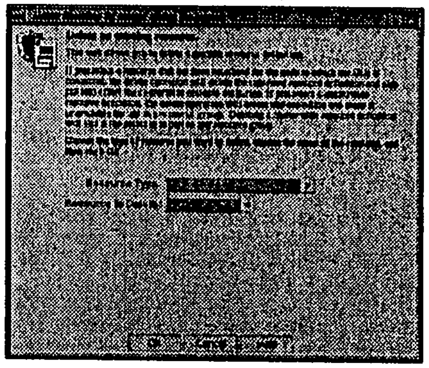
Sub-components:

- RichText
- TaskShelf
- ItemView
- ItemTable
- TreeView

Form

A Form is a single-page GUI for performing a Task. It contains a product-specific Task icon in the upper left corner, the Task title, and some introductory text describing the inputs the user is expected to type or choose. The Task-specific inputs themselves appear in the middle, with generic OK/Cancel/Help buttons at the bottom. All text in blue behaves like a hyperlink which launches glossary information in a separate small window.

The purpose of the Form interface is to make the entry of Task parameters simple and fast. It is suitable for Tasks of low complexity and a small number of parameters. Forms are the preferred interface when the typical users are knowledgeable and comfortable with the system being administered.



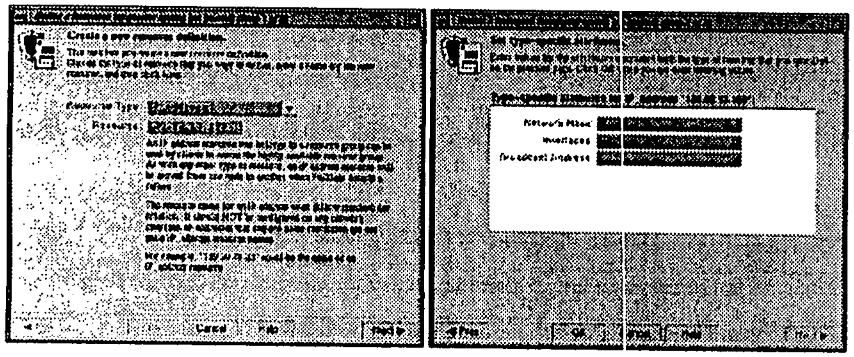
Guide

A Guide is a multi-page GUI containing explanatory text with a small set of labelled input components on each page. Like the Form, each page of the Guide contains a product-specific Task icon in the upper left corner, a title, and text describing the input expected of the user. The Task-specific inputs themselves appear in the middle, with generic Previous/Next/Cancel/Help buttons at the bottom. An "OK" button is presented when the user has navigated to the last page in the Guide. As in the Form, all text in blue behaves like a hyperlink which launches glossary information in a separate small window.

The purpose of the Guide interface is to provide step-by-step guidance on completing a complex task or a task with a large number of parameters. Guides are the preferred interface when the typical users are novices or not comfortable with the system being administered.

The first example shows page 1 of the "Define a Resource" Task. The "Prev" and "OK" buttons are automatically disabled; the descriptive text below the green input fields changes depending on what value is selected in the first field. The second example shows page 2 of the same Task; the buttons have been automatically enabled and disabled appropriately, and the labels and input fields have been determined by the values entered by the user on page 1.

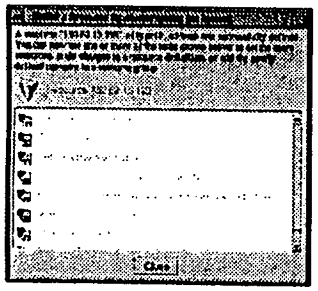
P58



ResultView

ResultView is a window which displays the results of a Task which has been successfully completed. (If the Task could not be performed, the user is given an error message describing the problem, and the Form or Guide remains open until the Task is successfully completed or explicitly cancelled.)

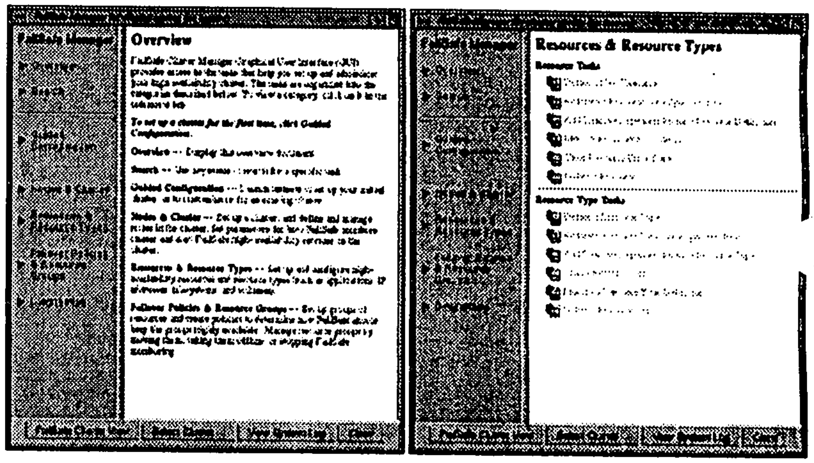
A ResultView contains a descriptive message, an icon representing the Item which was operated on (if applicable), and a TaskShelf showing the related Tasks which the user may want to launch next.



TaskManager

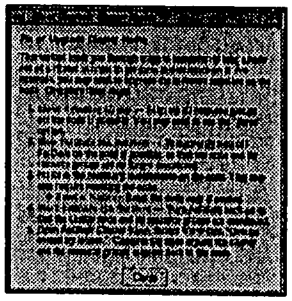
Task Manager is a front-end window that organizes an application's Tasks for easy access. Tasks are grouped into separate pages based on the types of Items that the Tasks operate upon, and the list of pages is presented as a table of contents. The Task Manager also includes an Overview and a Search page.

The first example shows the FailSafe 2.0 Cluster Manager GUI in its initial state; the Overview page is displayed. The second example shows the Tasks which are displayed when the user clicks on the "Resources & Resource Types" title.



MetaTask

A MetaTask is a logical combination of tasks that allows a user to achieve a high-level goal. In its simplest form, the MetaTask simply presents the list of Tasks along with a description of each, but allows the user to perform the Tasks in any order.



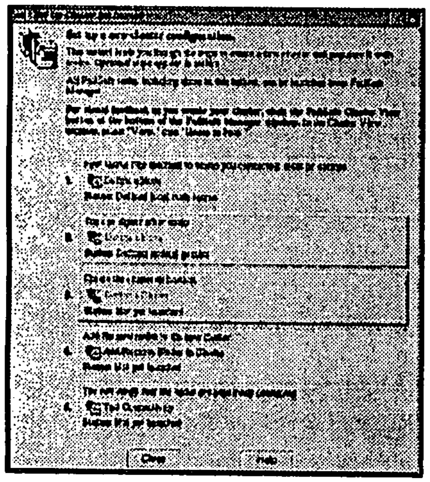
Smart MetaTask

A Smart MetaTask can be used by applications in which a series of Tasks must be performed in a specific order to reach a high-level goal. Smart MetaTasks can impose Task ordering and the passing of parameters from one Task to another. Specific Tasks can be disabled if the prerequisite conditions for performing the Task are not met.

P59

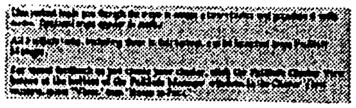
COPY

This example shows a Smart MetaTask containing five Tasks. The first is disabled because it's already been performed and should not be performed again. The second is optional and has already been performed, but it's still enabled because it can be performed any number of times. The third Task is enabled because it depended on the first Task, and the fourth and fifth Tasks are disabled because the third Task has not yet been completed.



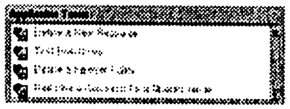
RichText

RichText is a text component that can display a small subset of HTML, including links. These links are most often used to bring up glossary definitions in another small window, but they can also be used to launch Tasks and other windows. Most of the Rhino components on this page contain one or more RichText components.



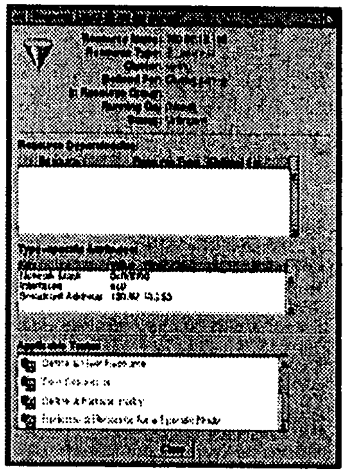
TaskShelf

A TaskShelf is a list of Tasks relevant to whatever GUI component contains the TaskShelf. The User can launch one of those Tasks by clicking on the Task name or icon. The TaskShelf is often dynamic, which means it will update the list of Tasks based on the state of the system.



ItemView

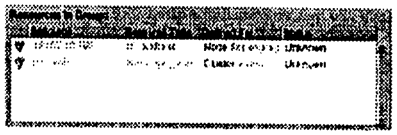
An ItemView shows all relevant information about an Item. The ItemView window displays simple key-value pairs at the top, application-specific contents such as ItemTables in the middle, and a TaskShelf at the bottom. The Item's icon is shown at the top left, and can change to reflect the I[illegible] state.



ItemTable

An ItemTable shows information about all Items in a Category. The user can sort on any column, and can click on the name of each Item to launch an ItemView containing the Item.

In this example, the other text in blue can also be clicked on to bring up other ItemViews.

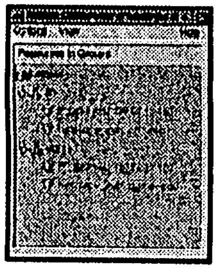


P60

TreeView

A TreeView displays Items that have a natural hierarchical relationship in an outline-style indented overview. Because it lets the user monitor the states of several Items at once, TreeView can be appropriate in a front-end monitoring window.

In this example from FailSafe 2.0 GUI, three different kinds of Items are shown; the cluster "winter" contains two resource groups, "phi" and "sigma," and each resource group contains two resources.



P61

Rhino Features

This lists some of the features and benefits of using Rhino.

Security

- **Secure connections**
  - Encrypted communication is supported (pending export compliance approval)
  - Security plugins such as SSH (secure shell) are supported
- **User login and authentication**
  - User must have an account on the server
  - GUI and server processes run as the user who logged in, not necessarily as root
- **Per-user privileges on tasks** - administrator can grant individual users permission to execute specific tasks
- **A single log file** records all system changes made by Rhino applications
- **Consistent security mechanism** - the same security practices are applicable to all Rhino applications

For more detail on the security model, see the Rhino Architecture document.

User Interfaces

- **Both GUI and non-GUI administration** - in addition to the GUI interface, the command-line interface allows administrators to write scripts
- **Glossary definitions** can be linked (like hypertext) to arbitrary text and label UI components
- Supports access to **end-user help**
  - As an application on IRIX, launches SGI Help
  - As an applet in browser, loads web page
- **Operates in a heterogeneous environment**
  - The Java clients run on IRIX, Linux, Windows, and under any browser that supports Java
  - The C++ server runs on IRIX and Linux, is being ported to UNICOS, and could be ported to Windows
- **Consistent GUI implementation**
  - All Rhino system administration tools look and behave similarly, which simplifies administration by reducing the learning curve and supporting consistent usage patterns
  - Implements proven user model and UI components which shipped in IRIX 6.3, 6.4, and 6.5, and were awarded the 1998 "Interaction Design Award"

For descriptions and screen shots of the high-level GUI components provided by Rhino, see the Rhino GUI Components document.

Monitoring

Rhino is good for applications which dynamically display the state of a system. These applications need to know when there has been a change originating outside of the application itself. Rhino provides an API that plug-ins can use to communicate state changes to the Rhino infrastructure; the infrastructure then notifies all GUIs that need to be updated. This allows system state to be accurately reflected in the GUI regardless of whether changes are caused by:

- A user running the GUI on another machine
- Another user executing command-line programs
- An administrator editing configuration files
- The system changing state by itself

Development features

- **Generates optional debug information**
  - Both client and server can produce debug messages
  - Infrastructure can print debug messages to help you fix implementation problems
  - Rhino provides API for putting debug messages in your code
  - Debug level can be set at runtime
- **Enables developers to focus on their problem domain and reduces development time** by providing the authentication, communication, and high-level GUI components which are common to administration applications
- **All clients benefit** from future infrastructure enhancements
- **Supports internationalization** by retrieving all GUI text from property files
- **Provides modular components.** Components from one Rhino application can be used in other Rhino applications. For example, a Disk Manager, Volume Manager, and Filesystem Manager could share components.
- Modular architecture allows additional functionality to be added to existing applications

P62

Basic Rhino Concepts

This is an introduction to basic concepts which will be used throughout the rest of the Rhino documentation.

Task

A **Task** is defined as *an atomic operation that changes the state of the system*. For example, Tasks that deal with user accounts might include "Add a User Account," "Modify a User Account," and "Change a User Account Password."

Each Task can be presented to the user through one (or both) of two interfaces: a terse, single-page **Form** or a more verbose, multi-page **Guide**. If both interfaces are defined for a Task, each will display a control that allows the user to switch between the two at any point while performing the Task.

Tasks can be launched from the command line, from an application or applet which uses a **Task Manager** (a GUI component which contains organized sets of Tasks), and from many other Rhino components. The execution of a Task invokes one or more privileged commands (described below) on the server.

Multiple Tasks may be combined into a **MetaTask**, which is a GUI component containing a sequence of steps leading to a high-level goal.

Item and Category

Item and Category are the mechanism by which the server tells the client about the state of the system.

An **Item** represents a physical or logical entity that is manipulated by system administration operations. Each Item has an associated type and a unique name within that type. For example, a user account named "foo" can be represented as an Item of type "user account" and unique name "foo".

A **Category** represents a dynamic collection of Items of a specific type. For example, the collection of user account Items can be represented as a Category.

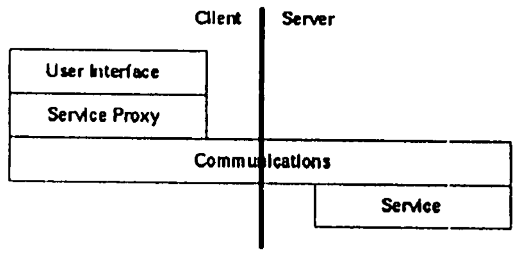
Privileged Command

A privileged command (or **"priv command"** for short) is a command-line program which is run on the server to change the state of the system. Priv commands are not setuid, but they are run by the `runpriv` command, which is setuid. `runpriv` checks privileges, makes a log entry, and runs the priv command as root.

Priv commands have been in use since IRIX 6.3.

P63

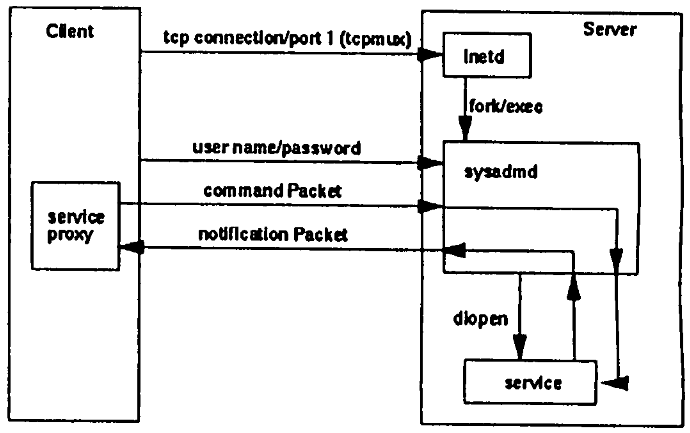
Rhino Architecture



The Rhino Architecture consists of three different interacting susbsystems. The Communications subsystem handles the transfer of data between client and server. The Services subsystem models the system to clients and provides a mechanism for making changes to the system. The Services subsystem is further divided into client (or service proxy) and server components. The User Interface subsystem provides a framework and components for developing System Administration user interfaces.

Communications

Note: The material in this section is provided so that developers can understand how the Rhino architecture works. Rhino developers should never have to interact explicitly with the Communications subsystem; instead, Rhino developers use the services described in the next section.



The Communications subsystem is responsible for transferring data back and forth between the client and the server.

The Rhino server is called sysadmd, and is typically started by the client via inetd. The client connects to port 1 on the server machine, which is serviced by inetd. The client makes a request to inetd that it start the "sgi_sysadm" tcpmux service, and inetd runs sysadmd.

In order for the client to do anything useful, it must first authenticate itself with the server. When sysadmd is started from inetd, the user must provide a valid user name/password combination. sysadmd will not respond to any requests other than authentication requests until a valid user name/password has been supplied.

When sysadmd is started by inetd, it is running as root. Once a valid user name/password is sp sysadmd sets its user id and its group id so that it is running with the permissions of the user na was specified. In this way, the system is protected from security problems with sysadmd because t user can't do anything via sysadmd that he or she could not do by logging into the system.

Once the user has been authenticated, communication between the client and the server takes the form of commands from the client to the server and notifications from the server to the client. The basic unit of communication is the Packet, and each Packet contains a type which identifies which service it is associated with and a selector which indicates which command or notification is being sent. Additionally, each Packet contains key/value pairs of information which specify any additional information needed to convey the command or notification.

sysadmd starts the sysadmd service at startup. The sysadmd service has commands that the client uses to load and unload other services. The client specifies in a Packet which service to be loaded, and sysadmd looks in /usr/sysadm/services to find the dynamic shared object (DSO) which implements the requested service. sysadmd dynamically loads the service, and henceforth any packets received by sysadmd having the type for that service get routed to the service's **handlePacket** method. Additionally, a service may send Packets back to a proxy running on the client. The client matches the type of a Packet from the server to the appropriate proxy and calls its **handlePacket** method.

Services

The Rhino Architecture provides four services that clients can use to access the server system. All services are available via the HostContext accessor methods getCategory, getAssociation, getTaskRegistry, and getPrivBroker. A HostContext instance is initially available to clients that have implemented RApp or RApplet subclasses, and is typically accessible in other contexts from a UIContext instance. Callers of HostContext methods do not need to be concerned with sending Packets or loading services. The HostContext accessor methods return service proxies which encapsulate all interactions with sysadmd.

Category Service

The Category service is used by clients to get information about the system. On the server side, a Category monitors some aspect of the system, and maintains an Item for each entity. The client is notified when Items are added, changed, or removed.

The Category Service is described in more detail in Item and Category in Rhino.

Association Service

The Association service maintains state representing relationships between Items on the system.

The Association Service is described in more detail in Item and Category in Rhino.

Task Registry Service

The Task Registry Service fetches lists of tasks from the server based on a variety of criteria.

Privilege Broker Service

The Privilege Broker Service lets the client run privileged commands on the server. This is the only way in which a Rhino client can make changes to the system.

PrivBroker provides a variety of ways in which the arguments may be specified to a privileged command. The runPriv method which passes arguments in the form of an AttrBundle is very convenient when the privileged command on the server uses *libsysadmParam* (see */usr/include/sysadm/SaParam.h*) to parse its command line arguments. The Privilege Broker service translates the AttrBundle into a format that is compatible with the parsing done by *libsysadmParam*. Since TaskData (see User Interface section below) is derived from AttrBundle, it is possible that the TaskData containing the parameters that the user has specified for a Task may be passed directly to the PrivBroker service.

See the *runpriv*(1M) man page for more information on the Irix privilege mechanism.

User Interface

The user interface subsystem consists of a few high-level framework components, along with many smaller components which can be used to build applications. The high-level framework components are:

- Task provides a user interface for making a change to the system.
- Task Manager organizes tasks into pages from which the user can launch them.
- Item View displays information about an administered item.
- RApp is the base class used for deriving new Rhino applications.
- RApplet is the base class used for deriving new Rhino applets.

The other components include:

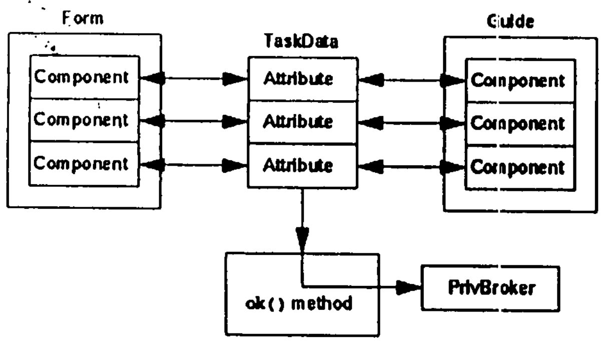
- ItemFinder is a JComboBox populated with Items from a Category or Association.
- ItemTable is a JTable populated with Items from a Category.
- EditableList provides a user interface for editing a list of entries.
- RichTextComponent displays formatted text.
- RichTextArea is a subclass of RichTextComponent that supports the display of glossary entries.
- RCheckBox, RButtonGroup, RDialog, RFrame, RLabel, RPanel, RPasswordField, RRadioButton, and RTextField are Rhino specializations of similarly named "J" Components from Swing.

The DynamicSize and DynamicSizeLayoutManager interfaces are the basis for Rhino dynamic geometry management. Rhino dynamic geometry management is implemented by Components whose heights depends on their widths, such as RichTextComponent. Dynamically sized Components implement the DynamicSize interface, which DynamicSizeLayoutManagers can use to determine the correct height to allocate for a Component given its width.

See the com.sgi.sysadm.ui, com.sgi.sysadm.ui.richText, com.sgi.sysadm.ui.event, and com.sgi.sysadm.ui.manager packages for complete listings of Rhino UI Components.

Task Architecture

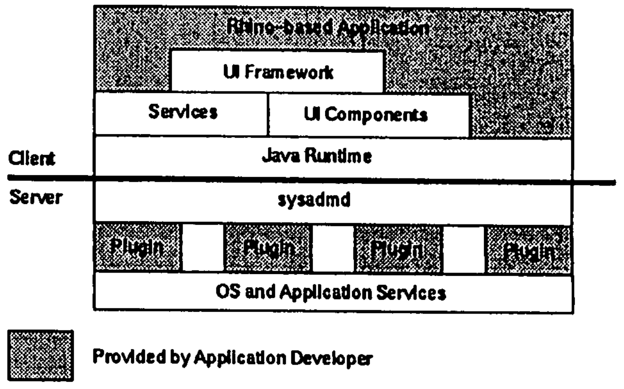
P65



A Task provides one or more user interfaces which prompt the user for parameters for making a change to the system, and an *ok* method that gets called when the user presses the OK button. Since a Task can have more than one user interface (Form and Guide), and since the user can switch back and forth between user interfaces, the TaskData mechanism is provided so that data is not lost when the user switches user interfaces.

The Task's internal representation of what the user has entered is stored as Attributes in the TaskData. Each Component in each of the user interfaces of a task is bound to an Attribute in the TaskData, so that when the Component changes, the TaskData is changed, and when the TaskData changes, the Component is changed. Thus, all input is preserved when the user switches back and forth between Form and Guide, and the *ok* method can get the parameters to pass to the Privilege Broker Service from the common TaskData rather than querying the user interface Components.

Architecture of a Rhino-based Application



The above diagram illustrates how the various pieces of the Rhino Architecture fit together with an application. On the client side, the application is in control, and uses the Rhino infrastructure to help implement its functionality. On the server side, sysadmd is in control, and it accesses developer-supplied plugins as requested by the client. A Plugin can take the form of a Category, Association, or Privileged Command, and it can also take the form of Task Registry entries.

P66